

Azure IoT Edge documentation

Azure IoT Edge extends IoT Hub. Analyze device data locally instead of in the cloud to send less data to the cloud, react to events quickly, and operate offline.

About IoT Edge development

OVERVIEW

[What is Azure IoT Edge?](#)

[What is Azure IoT Edge for Linux on Windows \(EFLOW\)?](#)

CONCEPT

[Understand the Azure IoT Edge runtime and its architecture](#)

[Azure IoT Edge versions and release notes](#)

[Azure IoT Edge supported systems](#)

HOW-TO GUIDE

[How an IoT Edge device can be used as a gateway](#)

Getting started using Linux devices

QUICKSTART

[Deploy your first IoT Edge module to a virtual Linux device](#)

TUTORIAL

[Develop Azure IoT Edge modules using Visual Studio Code](#)

HOW-TO GUIDE

[Install the IoT Edge runtime on Linux devices](#)

[Register and provision Linux devices at scale](#)

[Debug IoT Edge modules using Visual Studio Code](#)



REFERENCE

[IoT Edge open-source repository ↗](#)

Getting started using Windows devices



QUICKSTART

[Deploy your first IoT Edge module to a Windows device](#)



TUTORIAL

[Develop using IoT Edge for Linux on Windows](#)



HOW-TO GUIDE

[Create and provision IoT Edge for Linux on Windows](#)

[Install the IoT Edge runtime on Windows devices](#)



REFERENCE

[IoT Edge for Linux on Windows open-source repository ↗](#)

What is Azure IoT Edge

Article • 06/12/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge is a device-focused runtime that enables you to deploy, run, and monitor containerized Linux workloads.

Analytics drives business value in IoT solutions, but not all analytics need to be in the cloud. Azure IoT Edge helps you bring the analytical power of the cloud closer to your devices to drive better business insights and enable offline decision making. For example, you can run anomaly detection workloads at the edge to respond as quickly as possible to emergencies happening on a production line. If you want to reduce bandwidth costs and avoid transferring terabytes of raw data, you can clean and aggregate the data locally then only send the insights to the cloud for analysis.

Azure IoT Edge is a feature of [Azure IoT Hub](#) and enables you to scale out and manage an IoT solution from the cloud. By packaging your business logic into standard containers and using optional pre-built IoT Edge module images from partners or the [Microsoft Artifact Registry](#), you can easily compose, deploy, and maintain your solution.

Azure IoT Edge is made up of three components:

- **IoT Edge modules** are containers that run Azure services, third-party services, or your own code. Modules are deployed to IoT Edge devices and execute locally on those devices.
- The **IoT Edge runtime** runs on each IoT Edge device and manages the modules deployed to each device.
- A **cloud-based interface** enables you to remotely monitor and manage IoT Edge devices.

Note

Azure IoT Edge is available in the free and standard tier of IoT Hub. The free tier is for testing and evaluation only. For more information about the basic and standard tiers, see [How to choose the right IoT Hub tier](#).

IoT Edge modules

IoT Edge modules are units of execution, implemented as Docker-compatible containers, that run your business logic at the edge. Multiple modules can be configured to communicate with each other, creating a pipeline of data processing. You can develop custom modules or package certain Azure services into modules that provide insights offline and at the edge.

Artificial intelligence at the edge

Azure IoT Edge allows you to deploy complex event processing, machine learning, image recognition, and other high value AI without writing it in-house. Azure services like Azure Stream Analytics and Azure Machine Learning can all be run on-premises via Azure IoT Edge. You're not limited to Azure services, though. Anyone is able to create AI modules for your own use.

Bring your own code

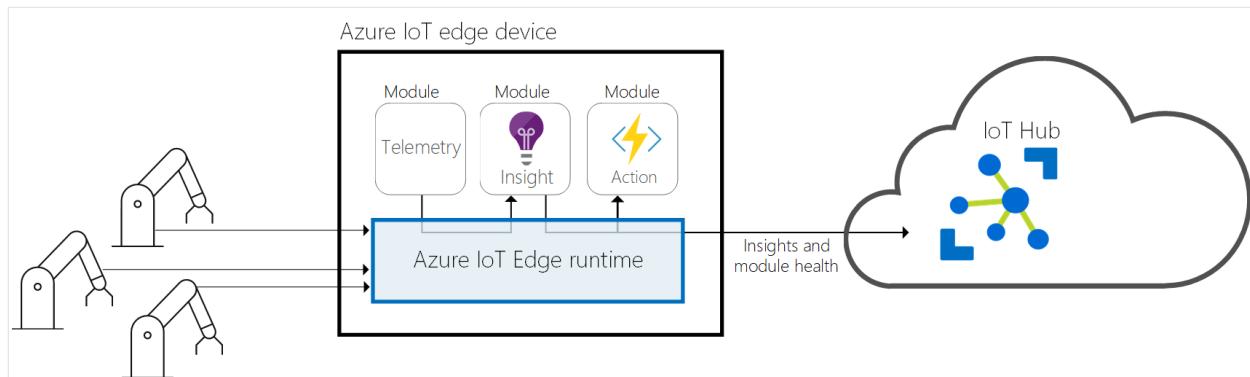
When you want to deploy your own code to your devices, Azure IoT Edge supports that, too. Azure IoT Edge holds to the same programming model as the other Azure IoT services. You can run the same code on a device or in the cloud. Azure IoT Edge supports both Linux and Windows so you can code to the platform of your choice. It supports Java, .NET Core 3.1, Node.js, C, and Python so your developers can code in a language they already know and use existing business logic.

IoT Edge runtime

The Azure IoT Edge runtime enables custom and cloud logic on IoT Edge devices. The runtime sits on the IoT Edge device, and performs management and communication operations. The runtime performs several functions:

- Installs and updates workloads on the device.
- Maintains Azure IoT Edge security standards on the device.
- Ensures that IoT Edge modules are always running.
- Reports module health to the cloud for remote monitoring.

- Manages communication between downstream devices and an IoT Edge device, between modules on an IoT Edge device, and between an IoT Edge device and the cloud.



How you use an Azure IoT Edge device is up to you. The runtime is often used to deploy AI to gateway devices which aggregate and process data from other on-premises devices, but this deployment model is just one option.

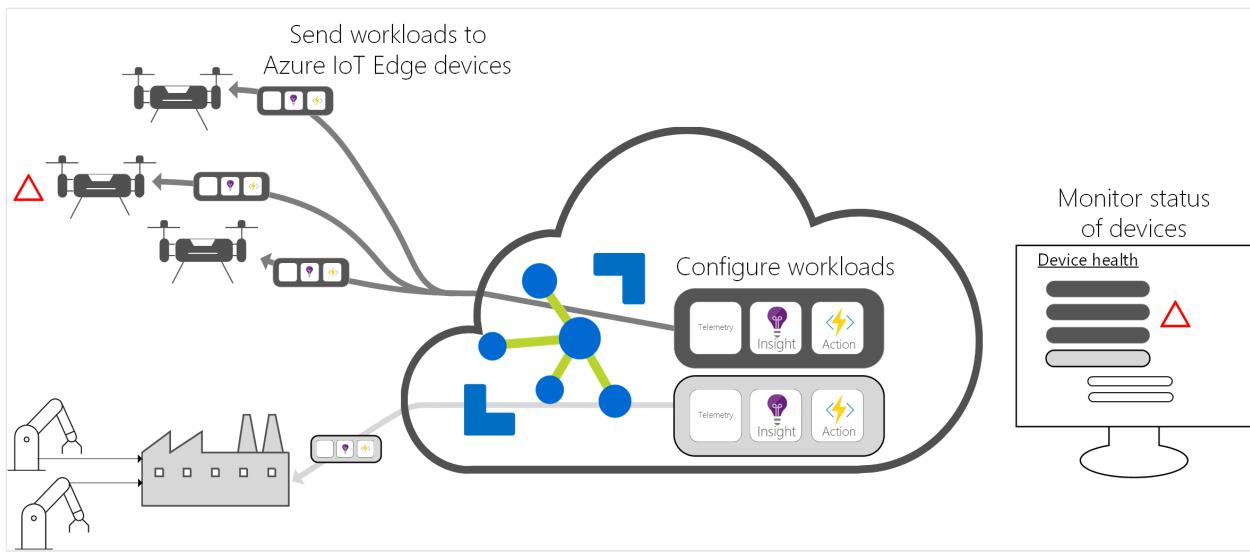
The Azure IoT Edge runtime runs on a large set of IoT devices that enables using it in a wide variety of ways. It supports both Linux and Windows operating systems and abstracts hardware details. Use a device smaller than a Raspberry Pi 3 if you're not processing much data, or use an industrial server to run resource-intensive workloads.

IoT Edge cloud interface

It's difficult to manage the software life cycle for millions of IoT devices that are often different makes and models or geographically scattered. Workloads are created and configured for a particular type of device, deployed to all of your devices, and monitored to catch any misbehaving devices. These activities can't be done on a per device basis and must be done at scale.

Azure IoT Edge integrates seamlessly with [Azure IoT Central](#) to provide one control plane for your solution's needs. Cloud services allow you to:

- Create and configure a workload to be run on a specific type of device.
- Send a workload to a set of devices.
- Monitor workloads running on devices in the field.



Next steps

Try out IoT Edge concepts by deploying your first IoT Edge module to a device:

- Deploy modules to a Linux IoT Edge device
- Deploy modules to a Windows IoT Edge device

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#)

Azure IoT Edge versions and release notes

Article • 05/01/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge is a product built from the open-source IoT Edge project hosted on GitHub. All new releases are made available in the [Azure IoT Edge project](#). Contributions and bug reports can be made on the [open-source IoT Edge project](#).

Azure IoT Edge is governed by Microsoft's [Modern Lifecycle Policy](#).

Documented versions

The IoT Edge documentation on this site is available for two different versions of the product. Currently, the two supported versions are:

- **IoT Edge 1.5 (LTS)** is the latest long-term support (LTS) version of IoT Edge and contains content for new features and capabilities that are in the latest stable release. The documentation for this version covers all features and capabilities from all previous versions through 1.5.
- **IoT Edge 1.4 (LTS)** is the previous long-term support (LTS) version of IoT Edge and is supported until November 12, 2024. This version of the documentation also contains content for the IoT Edge for Linux on Windows (EFLW). The documentation for this version is included with IoT Edge 1.5.

IoT Edge 1.1 (LTS) is the first long-term support (LTS) version of IoT Edge. It is no longer supported. The [documentation has been archived](#).

For more information about IoT Edge releases, see [Azure IoT Edge supported systems](#).

IoT Edge for Linux on Windows

Azure IoT Edge for Linux on Windows (EFLW) supports the following versions:

- **EFLOW Continuous Release (CR)** based on the latest non-LTS Azure IoT Edge version, it contains new features and capabilities that are in the latest stable release. For more information, see the [EFLOW release notes](#).
- **EFLOW 1.1 (LTS)** based on Azure IoT Edge 1.1, it's the Long-term support version. This version will be stable through the supported lifetime of this version and won't include new features released in later versions. This version will be supported until Dec 2022 to match the IoT Edge 1.1 LTS release lifecycle.
- **EFLOW 1.4 (LTS)** based on Azure IoT Edge 1.4. It's the latest Long-term support version. This version will be stable through the supported lifetime of this version and won't include new features released in later versions. This version will be supported until Nov 2024 to match the IoT Edge 1.4 LTS release lifecycle.

All new releases are made available in the [Azure IoT Edge for Linux on Windows project](#).

Version history

This table provides recent version history for IoT Edge package releases, and highlights documentation updates made for each version.

Note

Long-term servicing (LTS) releases are serviced for a fixed period. Updates to this release type contain critical security and bug fixes only. All other stable releases are continuously supported and serviced. A stable release may contain features updates along with critical security fixes. Stable releases are supported only until the next release (stable or LTS) is generally available.

 [Expand table](#)

Release notes and assets	Type	Release Date	End of Support Date	Highlights
1.5	Long-term support (LTS)	April 2024	November 10, 2026	IoT Edge 1.5 LTS is supported through November 10, 2026 to match the .NET 8 release lifecycle . Edge Agent and Edge Hub now support TLS 1.3 for inbound/outbound communication.
1.4	Long-term	August 2022	November 12, 2024	IoT Edge 1.4 LTS is supported through November 12, 2024 to match the .NET 6 release lifecycle .

Release notes and assets	Type	Release Date	End of Support Date	Highlights
	support (LTS)			<p>Automatic image clean-up of unused Docker images</p> <p>Ability to pass a custom JSON payload to DPS on provisioning</p> <p>Ability to require all modules in a deployment be downloaded before restart</p> <p>Use of the TCG TPM2 Software Stack which enables TPM hierarchy authorization values, specifying the TPM index at which to persist the DPS authentication key, and accommodating more TPM configurations</p>
1.3 ↗	Stable	June 2022	August 2022	<p>Support for Red Hat Enterprise Linux 8 on AMD and Intel 64-bit architectures.</p> <p>Edge Hub now enforces that inbound/outbound communication uses minimum TLS version 1.2 by default</p> <p>Updated runtime modules (edgeAgent, edgeHub) based on .NET 6</p>
1.2 ↗	Stable	April 2021	June 2022	<p>IoT Edge devices behind gateways</p> <p>IoT Edge MQTT broker (preview)</p> <p>New IoT Edge packages introduced, with new installation and configuration steps.</p> <p>Includes Microsoft Defender for IoT micro-agent for Edge.</p> <p>Integration with Device Update. For more information, see Update IoT Edge.</p>
1.1 ↗	Long-term support (LTS)	February 2021	December 13, 2022	<p>IoT Edge 1.1 LTS is supported through December 13, 2022 to match the .NET Core 3.1 release lifecycle.</p> <p>Long-term support plan and supported systems updates</p>
1.0.10 ↗	Stable	October 2020	February 2021	<p>UploadSupportBundle direct method</p> <p>Upload runtime metrics</p> <p>Route priority and time-to-live</p> <p>Module startup order</p> <p>X.509 manual provisioning</p>
1.0.9 ↗	Stable	March 2020	October 2020	<p>X.509 auto-provisioning with DPS</p> <p>RestartModule direct method</p> <p>support-bundle command</p>

IoT Edge for Linux on Windows

[+] Expand table

IoT Edge release	Available in EFLOW branch	Release date	End of Support Date	Highlights
1.4	Long-term support (LTS) ↗	November 2022	November 12, 2024	Azure IoT Edge 1.4.0 ↗ CBL-Mariner 2.0 ↗ USB passthrough using USB-Over-IP ↗ File/Folder sharing between Windows OS and the EFLOW VM ↗
1.3	Continuous release (CR) ↗	September 2022	In support	Azure IoT Edge 1.3.0 ↗ CBL-Mariner 2.0 ↗ USB passthrough using USB-Over-IP ↗ File/Folder sharing between Windows OS and the EFLOW VM ↗
1.2	Continuous release (CR) ↗	January 2022	September 2022	Public Preview ↗
1.1	Long-term support (LTS) ↗	June 2021	December 13, 2022	IoT Edge 1.1 LTS is supported through December 13, 2022 to match the .NET Core 3.1 release lifecycle ↗. Long-term support plan and supported systems updates

Next steps

- View all Azure IoT Edge releases ↗
- Make or review feature requests in the feedback forum ↗

Azure IoT Edge supported platforms

Article • 09/10/2024

Applies to:  IoT Edge 1.5

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article explains what operating system platforms, IoT Edge runtimes, container engines, and components are supported by IoT Edge whether generally available or in preview.

Get support

If you experience problems while using the Azure IoT Edge service, there are several ways to seek support. Try one of the following channels for support:

Reporting bugs - Most development that goes into the Azure IoT Edge product happens in the IoT Edge open-source project. Bugs can be reported on the [issues page](#) of the project. Bugs related to Azure IoT Edge for Linux on Windows can be reported on the [iotedge-eflow issues page](#). Fixes rapidly make their way from the projects in to product updates.

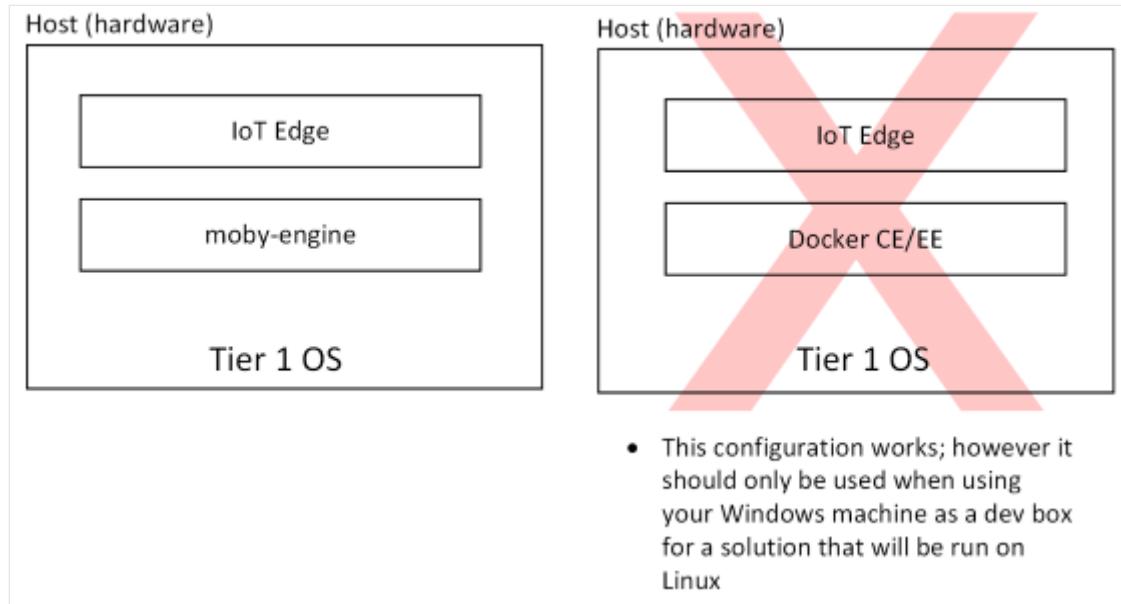
Microsoft Customer Support team - Users who have a [support plan](#) can engage the Microsoft Customer Support team by creating a support ticket directly from the [Azure portal](#).

Feature requests - The Azure IoT Edge product tracks feature requests via the product's [Azure feedback](#) community.

Container engines

Azure IoT Edge modules are implemented as containers, so IoT Edge needs a container engine to launch them. Microsoft provides a container engine, moby-engine, to fulfill this requirement. This container engine is based on the Moby open-source project. Docker CE and Docker EE are other popular container engines. They're also based on the Moby open-source project and are compatible with Azure IoT Edge. Microsoft provides best effort support for systems using those container engines; however, Microsoft can't

ship fixes for issues in them. For this reason, Microsoft recommends using moby-engine on production systems. If you are using Ubuntu Core snaps, the Docker snap is serviced by Canonical and supported for production scenarios.



Operating systems

Azure IoT Edge runs on most operating systems that can run containers; however, not all of these systems are equally supported. Operating systems are grouped into tiers that represent the level of support users can expect.

- Tier 1 systems are supported. For tier 1 systems, Microsoft:
 - has this operating system in automated tests
 - provides installation packages for them
- Tier 2 systems are compatible with Azure IoT Edge and can be used relatively easily. For tier 2 systems:
 - Microsoft has done informal testing on the platforms or knows of a partner successfully running Azure IoT Edge on the platform
 - Installation packages for other platforms may work on these platforms

Tier 1

The systems listed in the following tables are supported by Microsoft, either generally available or in public preview, and are tested with each new release.

Linux containers

Modules built as Linux containers can be deployed to either Linux or Windows devices. For Linux devices, the IoT Edge runtime is installed directly on the host device. For

Windows devices, a Linux virtual machine prebuilt with the IoT Edge runtime runs on the host device.

[IoT Edge for Linux on Windows](#) is the recommended way to run IoT Edge on Windows devices.

[+] [Expand table](#)

Operating System	AMD64	ARM32v7	ARM64	End of OS provider standard support
Debian 12 <small>↗</small>	✓			June 2028 <small>↗</small>
Debian 11 <small>↗</small>	✓			June 2026 <small>↗</small>
Red Hat Enterprise Linux 9 <small>↗</small>	✓			May 2032 <small>↗</small>
Red Hat Enterprise Linux 8 <small>↗</small>	✓			May 2029 <small>↗</small>
Ubuntu Server 24.04 <small>↗</small>	✓	✓		June 2029 <small>↗</small>
Ubuntu Server 22.04 <small>↗</small>	✓	✓		June 2027 <small>↗</small>
Ubuntu Server 20.04 <small>↗</small>	✓	✓		April 2025 <small>↗</small>
Ubuntu Core ¹ <small>↗</small>	✓	✓		April 2027 <small>↗</small>
Windows 10/11	✓	✓		See Azure IoT EFLW for supported Windows OS versions.
Windows Server 2019/2022	✓			See Azure IoT EFLW for supported Windows OS versions.

¹ Ubuntu Core is fully supported but the automated testing of Snaps currently happens on Ubuntu 22.04 Server LTS.

! Note

When a *Tier 1* operating system reaches its end of standard support date, it's removed from the *Tier 1* supported platform list. If you take no action, IoT Edge devices running on the unsupported operating system continue to work but ongoing security patches and bug fixes in the host packages for the operating system won't be available after the end of support date. To continue to receive support and security updates, we recommend that you update your host OS to a *Tier 1* supported platform.

Windows containers

We no longer support Windows containers. IoT Edge for Linux on Windows is the recommended way to run IoT Edge on Windows devices.

Tier 2

The systems listed in the following table are considered compatible with Azure IoT Edge, but aren't actively tested or maintained by Microsoft.

ⓘ Important

Support for these systems is best effort and may require you reproduce the issue on a tier 1 supported system.

[+] Expand table

Operating System	AMD64	ARM32v7	ARM64	End of OS provider standard support
Debian 12 ↗	✓		✓	June 2028 ↗
Debian 11 ↗	✓		✓	June 2026 ↗
Mentor Embedded Linux Flex OS ↗	✓	✓	✓	
Mentor Embedded Linux Omni OS ↗	✓		✓	
Ubuntu Server 24.04 ¹ ↗		✓		June 2029 ↗
Ubuntu Server 22.04 ¹ ↗		✓		June 2027 ↗
Ubuntu Server 20.04 ¹ ↗		✓		April 2025 ↗
Wind River 8 ↗	✓			
Yocto (scarthgap) ↗ For Yocto issues, open a GitHub issue ↗	✓	✓	✓	April 2028 ↗
Yocto (kirkstone) ↗ For Yocto issues, open a GitHub issue ↗	✓	✓	✓	April 2026 ↗

¹ Installation packages are made available on the [Azure IoT Edge releases](#). See the installation steps in [Offline or specific version installation](#).

ⓘ Note

When a *Tier 2* operating system reaches its end of standard support date, it's removed from the supported platform list. If you take no action, IoT Edge devices running on the unsupported operating system continue to work but ongoing security patches and bug fixes in the host packages for the operating system won't be available after the end of support date. To continue to receive support and security updates, we recommend that you update your host OS to a *Tier 1* supported platform.

Releases

The following table lists the currently supported releases. IoT Edge release assets and release notes are available on the [azure-iotedge releases](#) page.

ⓘ [Expand table](#)

Release notes and assets	Type	Release Date	End of Support Date
1.5	Long-term support (LTS)	April 2024	November 10, 2026
1.4	Long-term support (LTS)	August 2022	November 12, 2024

For more information on IoT Edge version history, see, [Version history](#).

ⓘ Important

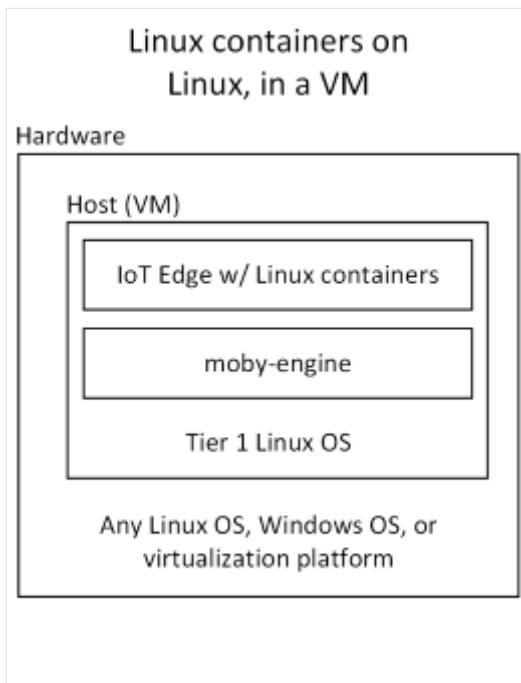
- Every Microsoft product has a lifecycle. The lifecycle begins when a product is released and ends when it's no longer supported. Knowing key dates in this lifecycle helps you make informed decisions about when to upgrade or make other changes to your software. IoT Edge is governed by Microsoft's [Modern Lifecycle Policy](#).

IoT Edge uses the Microsoft.Azure.Devices.Client SDK. For more information, see the [Azure IoT C# SDK GitHub repo](#) or the [Azure SDK for .NET reference content](#). The following list shows the version of the client SDK that each release is tested against:

IoT Edge version	Microsoft.Azure.Devices.Client SDK version
1.5	1.42.x
1.4	1.36.6

Virtual Machines

Azure IoT Edge can be run in virtual machines, such as an [Azure Virtual Machine](#). Using a virtual machine as an IoT Edge device is common when customers want to augment existing infrastructure with edge intelligence. The family of the host VM OS must match the family of the guest OS used inside a module's container. This requirement is the same as when Azure IoT Edge is run directly on a device. Azure IoT Edge is agnostic of the underlying virtualization technology and works in VMs powered by platforms like Hyper-V and vSphere.



Minimum system requirements

Azure IoT Edge runs great on devices as small as a Raspberry Pi3 to server grade hardware. Choosing the right hardware for your scenario depends on the workloads that you want to run. Making the final device decision can be complicated; however, you can easily start prototyping a solution on traditional laptops or desktops.

Experience while prototyping will help guide your final device selection. Questions you should consider include:

- How many modules are in your workload?
 - How many layers do your modules' containers share?
 - In what language are your modules written?
 - How much data will your modules be processing?
 - Do your modules need any specialized hardware for accelerating their workloads?
 - What are the desired performance characteristics of your solution?
 - What is your hardware budget?
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Quickstart: Deploy your first IoT Edge module to a virtual Linux device

Article • 07/08/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

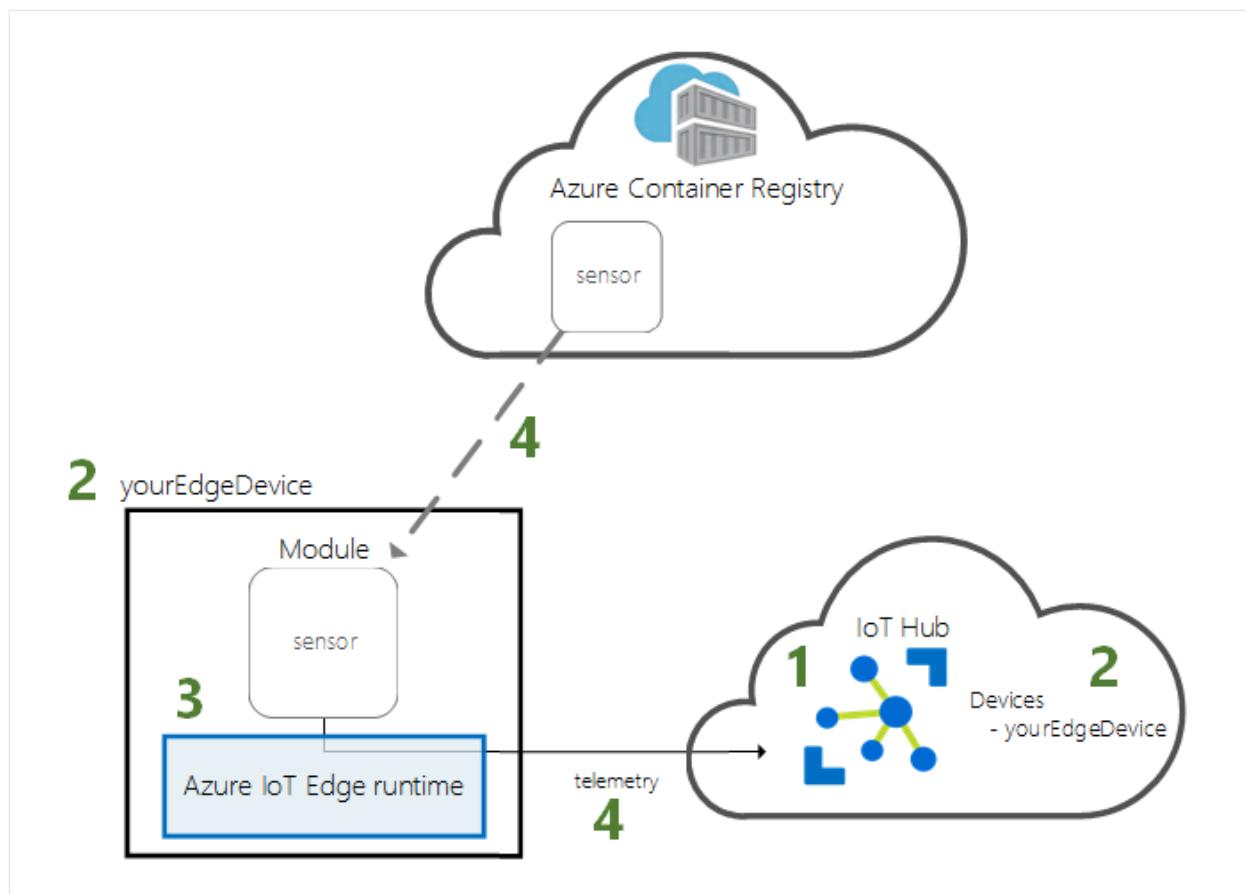
Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Test out Azure IoT Edge in this quickstart by deploying containerized code to a virtual Linux IoT Edge device. IoT Edge allows you to remotely manage code on your devices so that you can send more of your workloads to the edge. For this quickstart, we recommend using an Azure virtual machine for your IoT Edge device, which allows you to quickly create a test machine and then delete it when you're finished.

In this quickstart you learn how to:

- Create an IoT Hub.
- Register an IoT Edge device to your IoT hub.
- Install and start the IoT Edge runtime on a virtual device.
- Remotely deploy a module to an IoT Edge device.



This quickstart walks you through creating a Linux virtual machine that's configured to be an IoT Edge device. Then, you deploy a module from the Azure portal to your device. The module used in this quickstart is a simulated sensor that generates temperature, humidity, and pressure data. The other Azure IoT Edge tutorials build upon the work you do here by deploying additional modules that analyze the simulated data for business insights.

If you don't have an active Azure subscription, create a [free account](#) before you begin.

Prerequisites

Prepare your environment for the Azure CLI.

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).

[Launch Cloud Shell](#)

- If you prefer to run CLI reference commands locally, [Install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).

- If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).
- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
- Run [az version](#) to find the version and dependent libraries that are installed. To upgrade to the latest version, run [az upgrade](#).

Cloud resources:

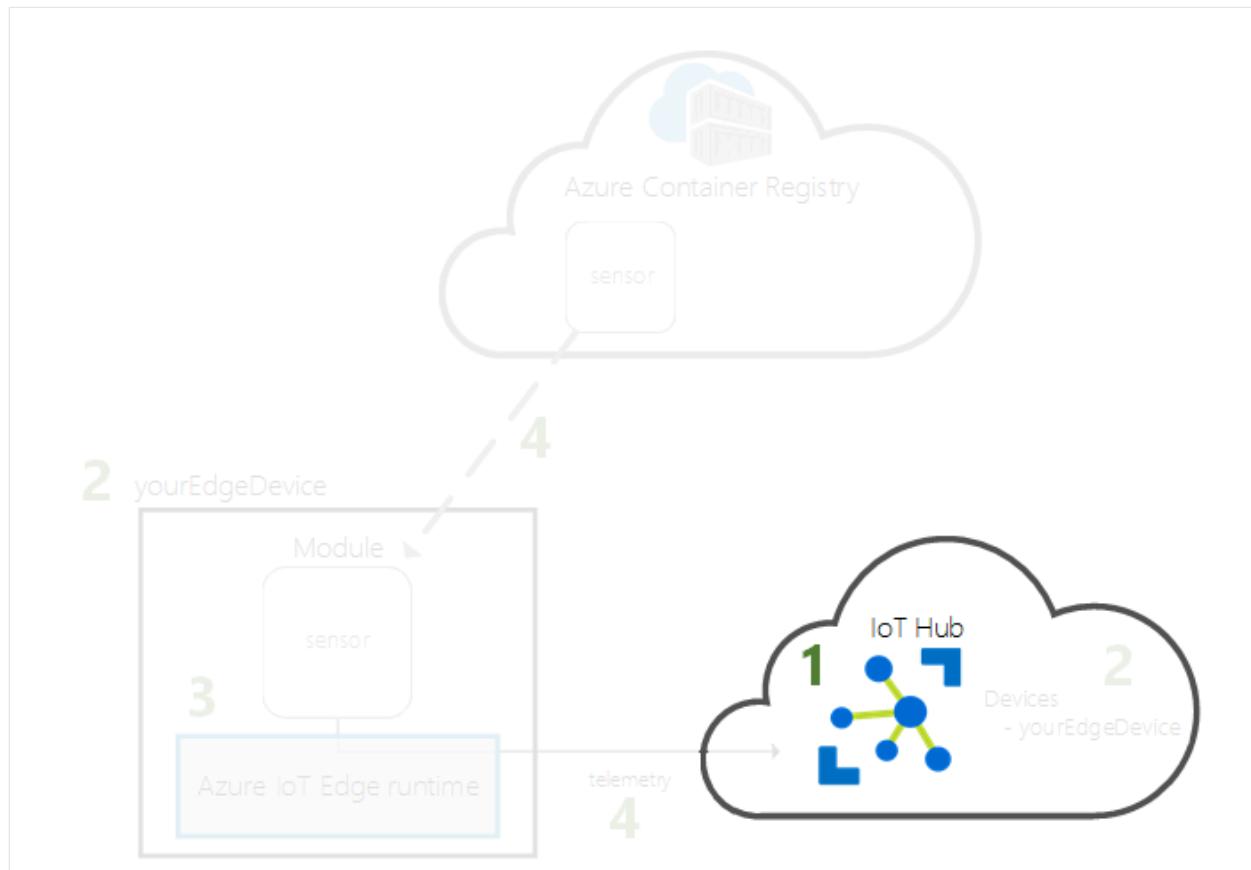
- A resource group to manage all the resources you use in this quickstart. We use the example resource group name **IoTEdgeResources** throughout this quickstart and the following tutorials.

Azure CLI

```
az group create --name IoTEdgeResources --location westus2
```

Create an IoT hub

Start the quickstart by creating an IoT hub with Azure CLI.



The free level of IoT Hub works for this quickstart. If you've used IoT Hub in the past and already have a hub created, you can use that IoT hub.

The following code creates a free F1 hub in the resource group **IoTEdgeResources**.

Replace `{hub_name}` with a unique name for your IoT hub. It might take a few minutes to create an IoT Hub.

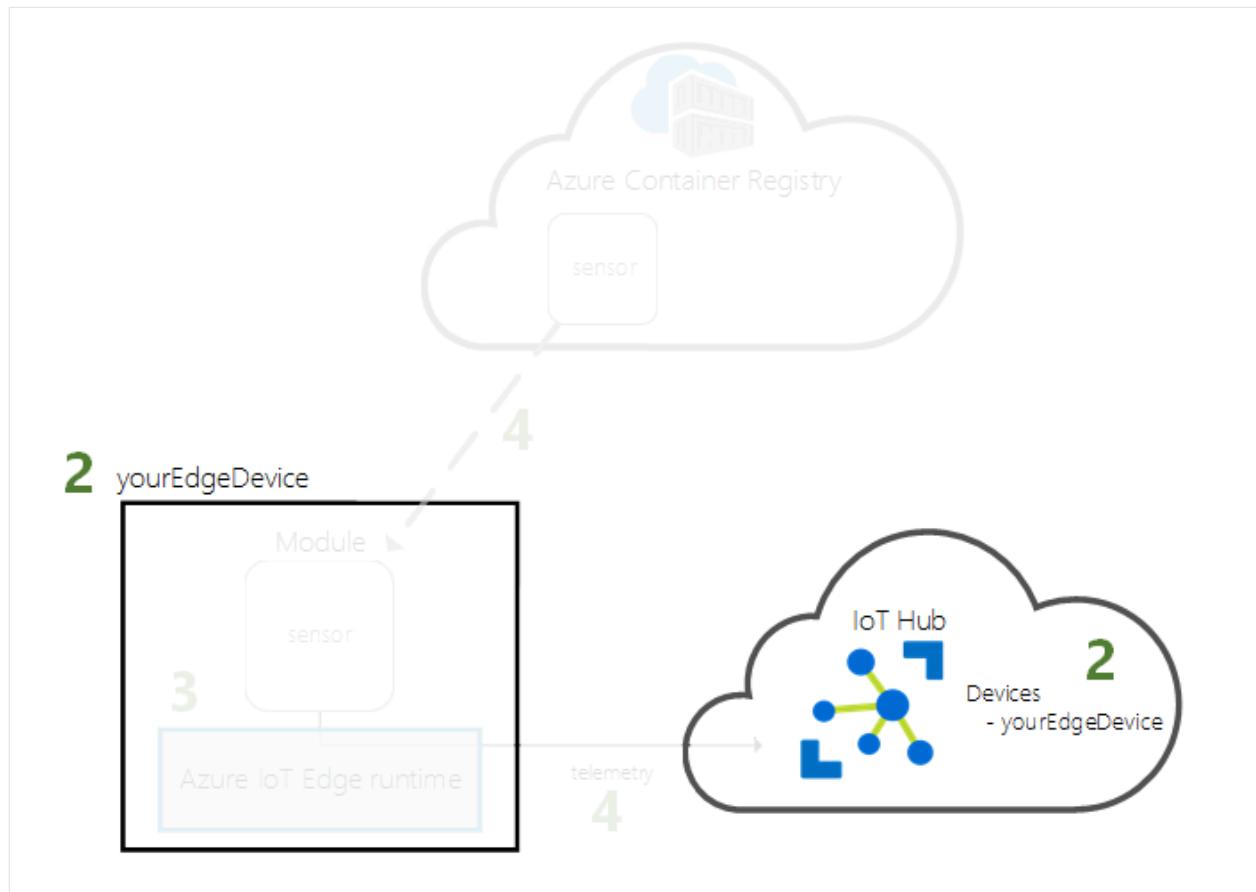
Azure CLI

```
az iot hub create --resource-group IoTEdgeResources --name {hub_name} --sku F1 --partition-count 2
```

If you get an error because there's already one free hub in your subscription, change the SKU to **S1**. Each subscription can only have one free IoT hub. If you get an error that the IoT Hub name isn't available, it means that someone else already has a hub with that name. Try a new name.

Register an IoT Edge device

Register an IoT Edge device with your newly created IoT hub.



Create a device identity for your IoT Edge device so that it can communicate with your IoT hub. The device identity lives in the cloud, and you use a unique device connection

string to associate a physical device to a device identity.

Since IoT Edge devices behave and can be managed differently than typical IoT devices, declare this identity to be for an IoT Edge device with the `--edge-enabled` flag.

1. In the Azure Cloud Shell, enter the following command to create a device named `myEdgeDevice` in your hub.

Azure CLI

```
az iot hub device-identity create --device-id myEdgeDevice --edge-enabled --hub-name {hub_name}
```

If you get an error about `iothubowner` policy keys, make sure that your Cloud Shell is running the latest version of the `azure-iot` extension.

2. View the connection string for your device, which links your physical device with its identity in IoT Hub. It contains the name of your IoT hub, the name of your device, and then a shared key that authenticates connections between the two. We'll refer to this connection string again in the next section when you set up your IoT Edge device.

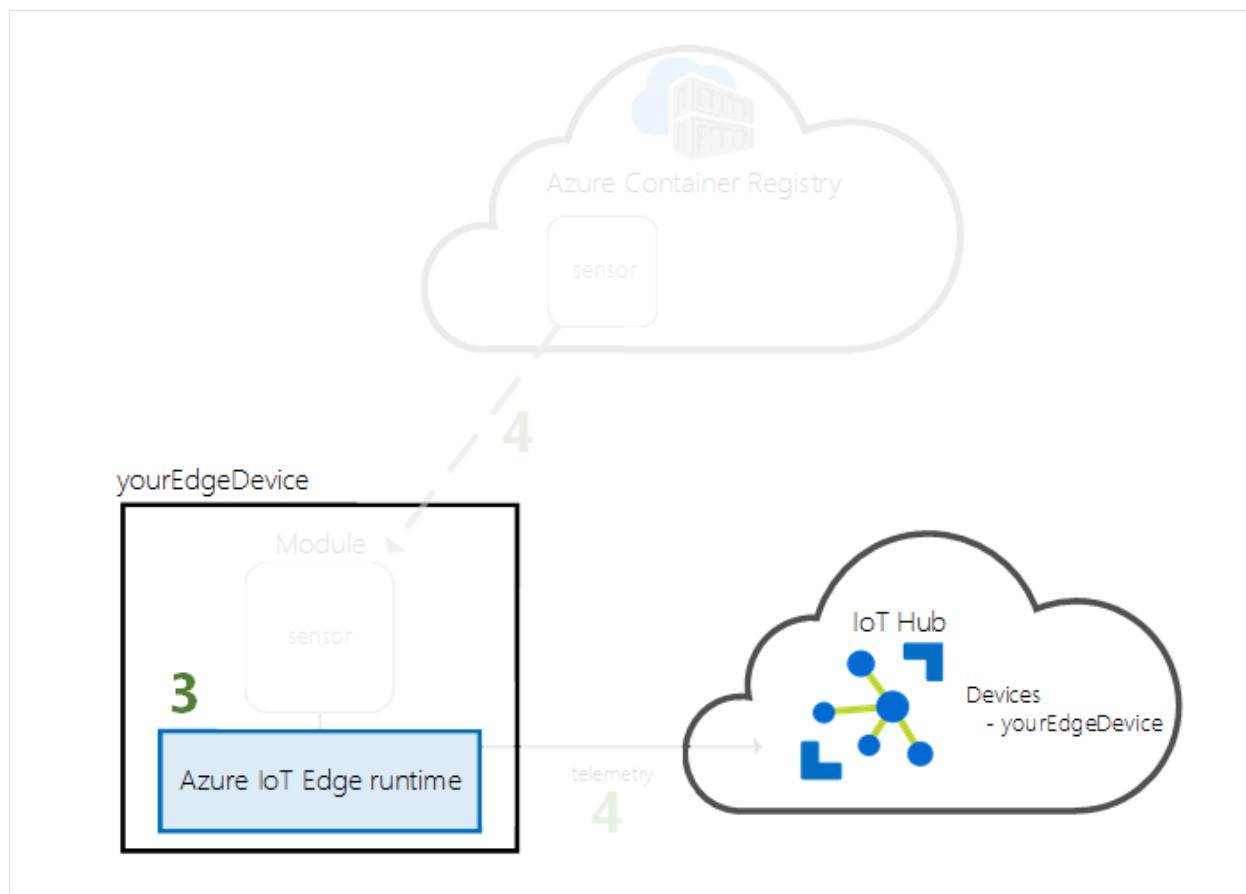
Azure CLI

```
az iot hub device-identity connection-string show --device-id myEdgeDevice --hub-name {hub_name}
```

For example, your connection string should look similar to `HostName=contoso-hub.azure-devices.net;DeviceId=myEdgeDevice;SharedAccessKey=<DEVICE_SHARED_ACCESS_KEY>`.

Configure your IoT Edge device

Create a virtual machine with the Azure IoT Edge runtime on it.



The IoT Edge runtime is deployed on all IoT Edge devices. It has three components. The *IoT Edge security daemon* starts each time an IoT Edge device boots and bootstraps the device by starting the IoT Edge agent. The *IoT Edge agent* facilitates deployment and monitoring of modules on the IoT Edge device, including the IoT Edge hub. The *IoT Edge hub* manages communications between modules on the IoT Edge device, and between the device and IoT Hub.

During the runtime configuration, you provide a device connection string. This is the string that you retrieved from the Azure CLI. This string associates your physical device with the IoT Edge device identity in Azure.

Deploy the IoT Edge device

This section uses an Azure Resource Manager template to create a new virtual machine and install the IoT Edge runtime on it. If you want to use your own Linux device instead, you can follow the installation steps in [Manually provision a single Linux IoT Edge device](#), then return to this quickstart.

Use the **Deploy to Azure** button or the CLI commands to create your IoT Edge device based on the prebuilt [iotedge-vm-deploy](#) template.

- Deploy using the IoT Edge Azure Resource Manager template.



- For bash or Cloud Shell users, copy the following command into a text editor, replace the placeholder text with your information, then copy into your bash or Cloud Shell window:

Azure CLI

```
az deployment group create \
--resource-group IoTEdgeResources \
--template-uri "https://raw.githubusercontent.com/Azure/iotedge-vm-
deploy/main/edgeDeploy.json" \
--parameters dnsLabelPrefix='<REPLACE_WITH_VM_NAME>' \
--parameters adminUsername='azureUser' \
--parameters deviceConnectionString=$(az iot hub device-identity
connection-string show --device-id myEdgeDevice --hub-name
<REPLACE_WITH_HUB_NAME> -o tsv) \
--parameters authenticationType='password' \
--parameters adminPasswordOrKey="<REPLACE_WITH_PASSWORD>"
```

- For PowerShell users, copy the following command into your PowerShell window, then replace the placeholder text with your own information:

Azure CLI

```
az deployment group create \
--resource-group IoTEdgeResources \
--template-uri "https://raw.githubusercontent.com/Azure/iotedge-vm-
deploy/main/edgeDeploy.json" \
--parameters dnsLabelPrefix='<REPLACE_WITH_VM_NAME>' \
--parameters adminUsername='azureUser' \
--parameters deviceConnectionString=$(az iot hub device-identity
connection-string show --device-id myEdgeDevice --hub-name
<REPLACE_WITH_HUB_NAME> -o tsv) \
--parameters authenticationType='password' \
--parameters adminPasswordOrKey="<REPLACE_WITH_PASSWORD>"
```

This template takes the following parameters:

[+] Expand table

Parameter	Description
resource-group	The resource group in which the resources will be created. Use the default IoTEdgeResources that we've been using throughout this article or provide the name of an existing resource group in your subscription.

Parameter	Description
template-uri	A pointer to the Resource Manager template that we're using.
dnsLabelPrefix	A string that will be used to create the virtual machine's hostname. Replace the placeholder text with a name for your virtual machine.
adminUsername	A username for the admin account of the virtual machine. Use the example azureUser or provide a new username.
deviceConnectionString	The connection string from the device identity in IoT Hub, which is used to configure the IoT Edge runtime on the virtual machine. The CLI command within this parameter grabs the connection string for you. Replace the placeholder text with your IoT hub name.
authenticationType	The authentication method for the admin account. This quickstart uses password authentication, but you can also set this parameter to sshPublicKey .
adminPasswordOrKey	The password or value of the SSH key for the admin account. Replace the placeholder text with a secure password. Your password must be at least 12 characters long and have three of four of the following: lowercase characters, uppercase characters, digits, and special characters.

Once the deployment is complete, you should receive JSON-formatted output in the CLI that contains the SSH information to connect to the virtual machine. Copy the value of the **public SSH** entry of the **outputs** section. For example, your SSH command should look similar to `ssh azureUser@edge-vm.westus2.cloudapp.azure.com`.

View the IoT Edge runtime status

The rest of the commands in this quickstart take place on your IoT Edge device itself, so that you can see what's happening on the device. If you're using a virtual machine, connect to that machine now using the admin username that you set up and the DNS name that was output by the deployment command. You can also find the DNS name on your virtual machine's overview page in the Azure portal. Use the following command to connect to your virtual machine. Replace `{admin username}` and `{DNS name}` with your own values.

Console

```
ssh {admin username}@{DNS name}
```

Once connected to your virtual machine, verify that the runtime was successfully installed and configured on your IoT Edge device.

1. Check to see that IoT Edge is running. The following command should return a status of **Ok** if IoT Edge is running, or provide any service errors.

```
Bash
```

```
sudo iotedge system status
```

 **Tip**

You need elevated privileges to run `iotedge` commands. Once you sign out of your machine and sign back in the first time after installing the IoT Edge runtime, your permissions are automatically updated. Until then, use `sudo` in front of the commands.

2. If you need to troubleshoot the service, retrieve the service logs.

```
Bash
```

```
sudo iotedge system logs
```

3. View all the modules running on your IoT Edge device. Since the service just started for the first time, you should only see the **edgeAgent** module running. The `edgeAgent` module runs by default and helps to install and start any additional modules that you deploy to your device.

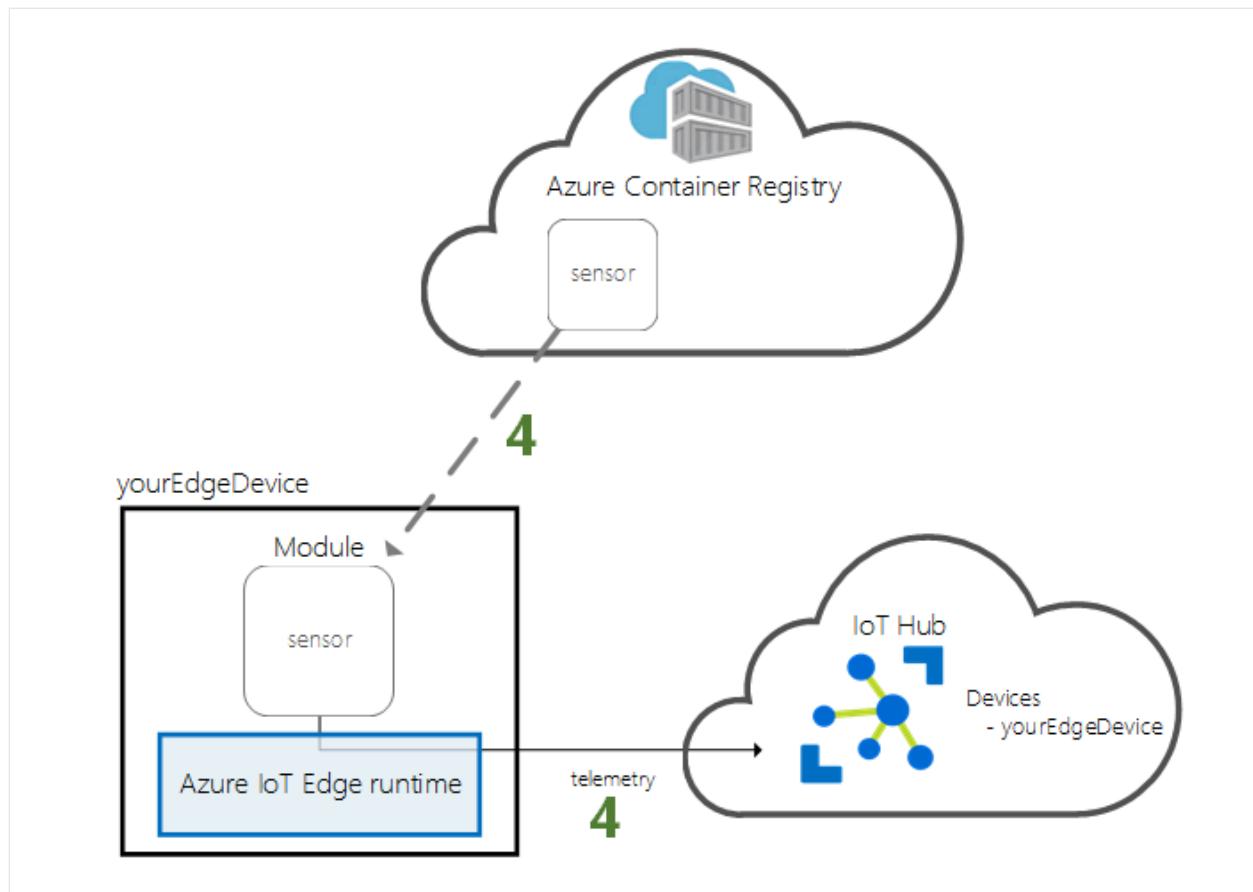
```
Bash
```

```
sudo iotedge list
```

Your IoT Edge device is now configured. It's ready to run cloud-deployed modules.

Deploy a module

Manage your Azure IoT Edge device from the cloud to deploy a module that will send telemetry data to IoT Hub.



One of the key capabilities of Azure IoT Edge is deploying code to your IoT Edge devices from the cloud. *IoT Edge modules* are executable packages implemented as containers. In this section, you'll deploy a pre-built module from the [IoT Edge Modules section of Microsoft Artifact Registry](#).

The module that you deploy in this section simulates a sensor and sends generated data. This module is a useful piece of code when you're getting started with IoT Edge because you can use the simulated data for development and testing. If you want to see exactly what this module does, you can view the [simulated temperature sensor source code](#).

Follow these steps to deploy your first module.

1. Sign in to the [Azure portal](#) and go to your IoT Hub.
2. From the menu on the left, under **Device Management**, select **Devices**.
3. Select the device ID of the target IoT Edge device from the list.

When you create a new IoT Edge device, it will display the status code `417 -- The device's deployment configuration is not set` in the Azure portal. This status is normal, and means that the device is ready to receive a module deployment.

4. On the upper bar, select **Set Modules**.

Choose which modules you want to run on your device. You can choose from modules that you've built yourself or images in a container registry. In this quickstart, you'll deploy a module from the Microsoft container registry.

5. In the **IoT Edge modules** section, select **Add** then choose **IoT Edge Module**.
6. Update the following module settings:

[+] Expand table

Setting	Value
IoT Module name	SimulatedTemperatureSensor
Image URI	mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:latest
Restart policy	always
Desired status	running

7. Select **Next: Routes** to continue to configure routes.
8. Add a route that sends all messages from the simulated temperature module to IoT Hub.

[+] Expand table

Setting	Value
Name	SimulatedTemperatureSensorToIoTHub
Value	FROM /messages/modules/SimulatedTemperatureSensor/* INTO \$upstream

9. Select **Next: Review + create**.
10. Review the JSON file, and then select **Create**. The JSON file defines all of the modules that you deploy to your IoT Edge device.

! **Note**

When you submit a new deployment to an IoT Edge device, nothing is pushed to your device. Instead, the device queries IoT Hub regularly for any new instructions. If the device finds an updated deployment manifest, it uses the information about the new deployment to pull the module images from the

cloud then starts running the modules locally. This process can take a few minutes.

After you create the module deployment details, the wizard returns you to the device details page. View the deployment status on the **Modules** tab.

You should see three modules: `$edgeAgent`, `$edgeHub`, and `SimulatedTemperatureSensor`. If one or more of the modules has **Yes** under **Specified in Deployment** but not under **Reported by Device**, your IoT Edge device is still starting them. Wait a few minutes, and then refresh the page.

Device Details						
Modules	IoT Edge hub connections	Deployments and Configurations	Logs	Metrics	Events	Power
Name	Type	Specified in Deployment	Reported by Device	Runtime Status	Exit Code	
<code>\$edgeAgent</code>	IoT Edge System Module	✓ Yes	✓ Yes	running	NA	
<code>\$edgeHub</code>	IoT Edge System Module	✓ Yes	✓ Yes	running	NA	
<code>SimulatedTemperatureSensor</code>	IoT Edge Custom Module	✓ Yes	✓ Yes	running	NA	

If you have issues deploying modules, see [Troubleshoot IoT Edge devices from the Azure portal](#).

View generated data

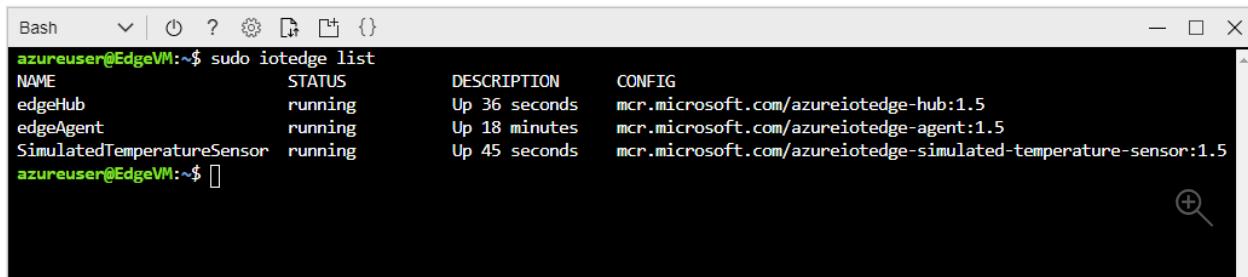
In this quickstart, you created a new IoT Edge device and installed the IoT Edge runtime on it. Then, you used the Azure portal to deploy an IoT Edge module to run on the device without having to make changes to the device itself.

In this case, the module that you pushed generates sample environment data that you can use for testing later. The simulated sensor is monitoring both a machine and the environment around the machine. For example, this sensor might be in a server room, on a factory floor, or on a wind turbine. The message includes ambient temperature and humidity, machine temperature and pressure, and a timestamp. The IoT Edge tutorials use the data created by this module as test data for analytics.

Open the command prompt on your IoT Edge device again, or use the SSH connection from Azure CLI. Confirm that the module deployed from the cloud is running on your IoT Edge device:

```
Bash
```

```
sudo iotedge list
```

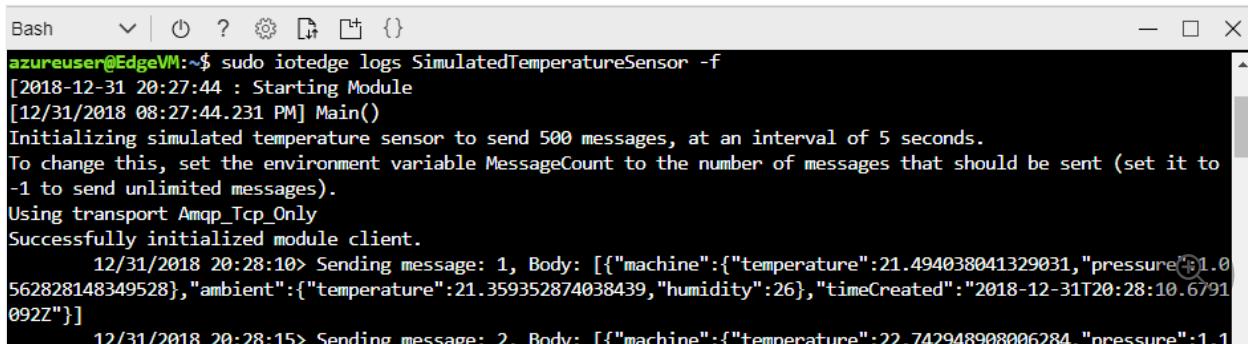


```
Bash azureuser@EdgeVM:~$ sudo iotedge list
NAME          STATUS    DESCRIPTION      CONFIG
edgeHub        running   Up 36 seconds  mcr.microsoft.com/azureiotedge-hub:1.5
edgeAgent       running   Up 18 minutes   mcr.microsoft.com/azureiotedge-agent:1.5
SimulatedTemperatureSensor  running   Up 45 seconds   mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.5
azureuser@EdgeVM:~$
```

View the messages being sent from the temperature sensor module:



```
Bash
sudo iotedge logs SimulatedTemperatureSensor -f
```



```
Bash azureuser@EdgeVM:~$ sudo iotedge logs SimulatedTemperatureSensor -f
[2018-12-31 20:27:44 : Starting Module
[12/31/2018 08:27:44.231 PM] Main()
Initializing simulated temperature sensor to send 500 messages, at an interval of 5 seconds.
To change this, set the environment variable MessageCount to the number of messages that should be sent (set it to -1 to send unlimited messages).
Using transport Amqp_Tcp_Only
Successfully initialized module client.

12/31/2018 20:28:10> Sending message: 1, Body: [{"machine": {"temperature": 21.494038041329031, "pressure": 1.056282148349528}, "ambient": {"temperature": 21.359352874038439, "humidity": 26}, "timeCreated": "2018-12-31T20:28:10.6791092Z"}]
12/31/2018 20:28:15> Sending message: 2, Body: [{"machine": {"temperature": 22.742948908006284, "pressure": 1.115625}, "ambient": {"temperature": 22.615625, "humidity": 26}, "timeCreated": "2018-12-31T20:28:15.742948908006284Z"}]
```

💡 Tip

IoT Edge commands are case-sensitive when referring to module names.

Clean up resources

If you want to continue on to the IoT Edge tutorials, you can use the device that you registered and set up in this quickstart. Otherwise, you can delete the Azure resources that you created to avoid charges.

If you created your virtual machine and IoT hub in a new resource group, you can delete that group and all the associated resources. Double check the contents of the resource group to make sure that there's nothing you want to keep. If you don't want to delete the whole group, you can delete individual resources instead.

ⓘ Important

Deleting a resource group is irreversible.

Remove the **IoTEdgeResources** group. It might take a few minutes to delete a resource group.

Azure CLI

```
az group delete --name IoTEdgeResources --yes
```

You can confirm the resource group is removed by viewing the list of resource groups.

Azure CLI

```
az group list
```

Next steps

In this quickstart, you created an IoT Edge device and used the Azure IoT Edge cloud interface to deploy code onto the device. Now, you have a test device generating raw data about its environment.

In the next tutorial, you'll learn how to monitor the activity and health of your device from the Azure portal.

[Monitor IoT Edge devices](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) ↗

Tutorial: Develop IoT Edge modules using Visual Studio Code

Article • 03/27/2024

ⓘ AI-assisted content. This article was partially created with the help of AI. An author reviewed and revised the content as needed. [Learn more](#)

Applies to:  IoT Edge 1.5  IoT Edge 1.4

ⓘ Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This tutorial walks through developing and deploying your own code to an IoT Edge device. You can use Azure IoT Edge modules to deploy code that implements your business logic directly to your IoT Edge devices. In the [Deploy code to a Linux device](#) quickstart, you created an IoT Edge device and deployed a module from the Azure Marketplace.

This article includes steps for two IoT Edge development tools.

- *Azure IoT Edge Dev Tool* command-line (CLI). This tool is preferred for development.
- *Azure IoT Edge tools for Visual Studio Code* extension. The extension is in [maintenance mode](#).

Use the tool selector button at the beginning of this article to select the tool version.

In this tutorial, you learn how to:

- ✓ Set up your development machine.
- ✓ Use the IoT Edge tools to create a new project.
- ✓ Build your project as a [Docker container](#) and store it in an Azure container registry.
- ✓ Deploy your code to an IoT Edge device.

The IoT Edge module that you create in this tutorial filters the temperature data that your device generates. It only sends messages upstream if the temperature is above a specified threshold. This type of analysis at the edge is useful for reducing the amount of data that's communicated to and stored in the cloud.

Prerequisites

A development machine:

- Use your own computer or a virtual machine.
- Your development machine must support [nested virtualization](#) for running a container engine.
- Most operating systems that can run a container engine can be used to develop IoT Edge modules for Linux devices. This tutorial uses a Windows computer, but points out known differences on macOS or Linux.
- Install [Visual Studio Code](#) ↗
- Install the [Azure CLI](#).

An Azure IoT Edge device:

- You should run IoT Edge on a separate device. This distinction between development machine and IoT Edge device simulates a true deployment scenario and helps keep the different concepts separate. Use the quickstart article [Deploy code to a Linux Device](#) to create an IoT Edge device in Azure or the [Azure Resource Template to deploy an IoT Edge enabled VM](#) ↗.

Cloud resources:

- A free or standard-tier [IoT hub](#) in Azure.

If you don't have an [Azure subscription](#), create an [Azure free account](#) ↗ before you begin.

💡 Tip

For guidance on interactive debugging in Visual Studio Code or Visual Studio 2022:

- [Debug Azure IoT Edge modules using Visual Studio Code](#)
- [Use Visual Studio 2022 to develop and debug modules for Azure IoT Edge](#)

This tutorial teaches the development steps for Visual Studio Code.

Key concepts

This tutorial walks through the development of an IoT Edge module. An *IoT Edge module* is a container with executable code. You can deploy one or more modules to an IoT Edge device. Modules perform specific tasks like ingesting data from sensors, cleaning

and analyzing data, or sending messages to an IoT hub. For more information, see [Understand Azure IoT Edge modules](#).

When developing IoT Edge modules, it's important to understand the difference between the development machine and the target IoT Edge device where the module deploys. The container that you build to hold your module code must match the operating system (OS) of the *target device*. For example, the most common scenario is someone developing a module on a Windows computer intending to target a Linux device running IoT Edge. In that case, the container operating system would be Linux. As you go through this tutorial, keep in mind the difference between the *development machine OS* and the *container OS*.

 **Tip**

If you're using [IoT Edge for Linux on Windows](#), then the *target device* in your scenario is the Linux virtual machine, not the Windows host.

This tutorial targets devices running IoT Edge with Linux containers. You can use your preferred operating system as long as your development machine runs Linux containers. We recommend using Visual Studio Code to develop with Linux containers, so that's what this tutorial uses. You can use Visual Studio as well, although there are differences in support between the two tools.

The following table lists the supported development scenarios for **Linux containers** in Visual Studio Code and Visual Studio.

 [Expand table](#)

	Visual Studio Code	Visual Studio 2019/2022
Linux device architecture	Linux AMD64 Linux ARM32v7 Linux ARM64	Linux AMD64 Linux ARM32 Linux ARM64
Azure services	Azure Functions Azure Stream Analytics Azure Machine Learning	
Languages	C C# Java Node.js Python	C C#

	Visual Studio Code	Visual Studio 2019/2022
More information	Azure IoT Edge for Visual Studio Code	Azure IoT Edge Tools for Visual Studio 2019 Azure IoT Edge Tools for Visual Studio 2022

Install container engine

IoT Edge modules are packaged as containers, so you need a [Docker compatible container management system](#) on your development machine to build and manage them. We recommend Docker Desktop for development because of its feature support and popularity. Docker Desktop on Windows lets you switch between Linux containers and Windows containers so that you can develop modules for different types of IoT Edge devices.

Use the Docker documentation to install on your development machine:

- [Install Docker Desktop for Windows](#)
 - When you install Docker Desktop for Windows, you're asked whether you want to use Linux or Windows containers. You can change this decision at any time. For this tutorial, we use Linux containers because our modules are targeting Linux devices. For more information, see [Switch between Windows and Linux containers](#).
- [Install Docker Desktop for Mac](#)
- Read [About Docker CE](#) for installation information on several Linux platforms.
 - For the Windows Subsystem for Linux (WSL), install Docker Desktop for Windows.

Set up tools

Install the Python-based [Azure IoT Edge Dev Tool](#) to create your IoT Edge solution. There are two options:

- Use the prebuilt [IoT Edge Dev Container](#)
- Install the tool using the [iotedge dev](#) setup

Install language specific tools

Install tools specific to the language you're developing in:

- [.NET Core SDK](#)
- [C# Visual Studio Code extension](#)

Create a container registry

In this tutorial, you use the [Azure IoT Edge](#) and [Azure IoT Hub](#) extensions to build a module and create a **container image** from the files. Then you push this image to a **registry** that stores and manages your images. Finally, you deploy your image from your registry to run on your IoT Edge device.

 **Important**

The Azure IoT Edge Visual Studio Code extension is in [maintenance mode](#).

You can use any Docker-compatible registry to hold your container images. Two popular Docker registry services are [Azure Container Registry](#) and [Docker Hub](#). This tutorial uses Azure Container Registry.

If you don't already have a container registry, follow these steps to create a new one in Azure:

1. In the [Azure portal](#), select **Create a resource > Containers > Container Registry**.
2. Provide the following required values to create your container registry:

 [Expand table](#)

Field	Value
Subscription	Select a subscription from the drop-down list.
Resource group	Use the same resource group for all of the test resources that you create during the IoT Edge quickstarts and tutorials. For example, IoTEdgeResources .
Registry name	Provide a unique name.
Location	Choose a location close to you.
SKU	Select Basic .

3. Select **Review + create**, then **Create**.
4. Select your new container registry from the **Resources** section of your Azure portal home page to open it.
5. In the left pane of your container registry, select **Access keys** from the menu located under **Settings**.

The screenshot shows the 'Access keys' configuration for a container registry named 'myRegistry'. The 'Enabled' toggle switch is set to 'Enabled' (indicated by a blue circle). The 'Admin user' section displays a generated 'Username' ('myusername') and a 'Name' ('password'). Below these, there are two 'Password' fields with regenerated values ('igY/pYHu6I6QOGP56V3p...' and 'FKfYcP42n+DEuhuxhHd...'). The left sidebar has a red box around the 'Access keys' link under the 'Settings' category.

6. Enable **Admin user** with the toggle button and view the **Username** and **Password** for your container registry.
7. Copy the values for **Login server**, **Username**, and **password** and save them somewhere convenient. You use these values throughout this tutorial to provide access to the container registry.

Create a new module project

The Azure IoT Edge extension offers project templates for all supported IoT Edge module languages in Visual Studio Code. These templates have all the files and code that you need to deploy a working module to test IoT Edge, or give you a starting point to customize the template with your own business logic.

Create a project template

The [IoT Edge Dev Tool](#) simplifies Azure IoT Edge development to commands driven by environment variables. It gets you started with IoT Edge development with the IoT Edge Dev Container and IoT Edge solution scaffolding that has a default module and all the required configuration files.

1. Create a directory for your solution with the path of your choice. Change into your `iotedgesolution` directory.

```
Bash
```

```
mkdir c:\dev\iotedgesolution  
cd c:\dev\iotedgesolution
```

2. Use the **iotedge dev solution init** command to create a solution and set up your Azure IoT Hub in the development language of your choice.

```
C#
```

```
Bash
```

```
iotedge dev solution init --template csharp
```

The *iotedge dev solution init* script prompts you to complete several steps including:

- Authenticate to Azure
- Choose an Azure subscription
- Choose or create a resource group
- Choose or create an Azure IoT Hub
- Choose or create an Azure IoT Edge device

After solution creation, these main files are in the solution:

- A `.vscode` folder contains configuration file `launch.json`.
- A **modules** folder that has subfolders for each module. Within the subfolder for each module, the `module.json` file controls how modules are built and deployed.
- An `.env` file lists your environment variables. The environment variable for the container registry is `localhost:5000` by default.
- Two module deployment files named `deployment.template.json` and `deployment.debug.template.json` list the modules to deploy to your device. By default, the list includes the IoT Edge system modules (`edgeAgent` and `edgeHub`) and sample modules such as:
 - **filtermodule** is a sample module that implements a simple filter function.
 - **SimulatedTemperatureSensor** module that simulates data you can use for testing. For more information about how deployment manifests work, see [Learn how to use deployment manifests to deploy modules and establish routes](#). For

more information on how the simulated temperature module works, see the [SimulatedTemperatureSensor.csproj source code ↗](#).

ⓘ Note

The exact modules installed may depend on your language of choice.

Set IoT Edge runtime version

The latest stable IoT Edge system module version is 1.5. Set your system modules to version 1.5.

1. In Visual Studio Code, open **deployment.template.json** deployment manifest file. The [deployment manifest](#) is a JSON document that describes the modules to be configured on the targeted IoT Edge device.
2. Change the runtime version for the system runtime module images **edgeAgent** and **edgeHub**. For example, if you want to use the IoT Edge runtime version 1.5, change the following lines in the deployment manifest file:

JSON

```
"systemModules": {  
    "edgeAgent": {  
  
        "image": "mcr.microsoft.com/azureiotedge-agent:1.5",  
  
    "edgeHub": {  
  
        "image": "mcr.microsoft.com/azureiotedge-hub:1.5",  
    }  
}}
```

Target architecture

You need to select the architecture you're targeting with each solution, because that affects how the container is built and runs. The default is Linux AMD64. For this tutorial, we're using an Ubuntu virtual machine as the IoT Edge device and keep the default **amd64**.

If you need to change the target architecture for your solution, use the following steps.

C#

The target architecture is set when you create the container image in a later step.

Update module with custom code

Each template includes sample code that takes simulated sensor data from the **SimulatedTemperatureSensor** module and routes it to the IoT hub. The sample module receives messages and then passes them on. The pipeline functionality demonstrates an important concept in IoT Edge, which is how modules communicate with each other.

Each module can have multiple *input* and *output* queues declared in their code. The IoT Edge hub running on the device routes messages from the output of one module into the input of one or more modules. The specific code for declaring inputs and outputs varies between languages, but the concept is the same across all modules. For more information about routing between modules, see [Declare routes](#).

C#

The sample C# code that comes with the project template uses the [ModuleClient Class](#) from the IoT Hub SDK for .NET.

1. In the Visual Studio Code explorer, open **modules > filtermodule > ModuleBackgroundService.cs**.
2. Before the **filtermodule** namespace, add three **using** statements for types that are used later:

C#

```
using System.Collections.Generic;      // For KeyValuePair<>
using Microsoft.Azure.Devices.Shared; // For TwinCollection
using Newtonsoft.Json;                // For JsonConvert
```

3. Add the **temperatureThreshold** variable to the **ModuleBackgroundService** class. This variable sets the value that the measured temperature must exceed for the data to be sent to the IoT hub.

C#

```
static int temperatureThreshold { get; set; } = 25;
```

4. Add the **MessageBody**, **Machine**, and **Ambient** classes. These classes define the expected schema for the body of incoming messages.

C#

```
class MessageBody
{
    public Machine machine {get;set;}
    public Ambient ambient {get; set;}
    public string timeCreated {get; set;}
}
class Machine
{
    public double temperature {get; set;}
    public double pressure {get; set;}
}
class Ambient
{
    public double temperature {get; set;}
    public int humidity {get; set;}
}
```

5. Find the **ExecuteAsync** function. This function creates and configures a **ModuleClient** object that allows the module to connect to the local Azure IoT Edge runtime to send and receive messages. After creating the **ModuleClient**, the code reads the **temperatureThreshold** value from the module twin's desired properties. The code registers a callback to receive messages from an IoT Edge hub via an endpoint called **input1**.

Replace the call to the **ProcessMessageAsync** method with a new one that updates the name of the endpoint and the method that's called when input arrives. Also, add a **SetDesiredPropertyUpdateCallbackAsync** method for updates to the desired properties. To make this change, replace the last line of the **ExecuteAsync** method with the following code:

C#

```
// Register a callback for messages that are received by the
// module.
// await _moduleClient.SetInputMessageHandlerAsync("input1",
// PipeMessage, cancellationToken);

// Read the TemperatureThreshold value from the module twin's
// desired properties
var moduleTwin = await _moduleClient.GetTwinAsync();
await OnDesiredPropertiesUpdate(moduleTwin.Properties.Desired,
    _moduleClient);

// Attach a callback for updates to the module twin's desired
// properties.
await
    _moduleClient.SetDesiredPropertyUpdateCallbackAsync(OnDesiredProper
```

```

tiesUpdate, null);

// Register a callback for messages that are received by the
// module. Messages received on the inputFromSensor endpoint are sent
// to the FilterMessages method.
await _moduleClient.SetInputMessageHandlerAsync("inputFromSensor",
FilterMessages, _moduleClient);

```

6. Add the `onDesiredPropertiesUpdate` method to the `ModuleBackgroundService` class. This method receives updates on the desired properties from the module twin, and updates the `temperatureThreshold` variable to match. All modules have their own module twin, which lets you configure the code that's running inside a module directly from the cloud.

C#

```

static Task OnDesiredPropertiesUpdate(TwinCollection
desiredProperties, object userContext)
{
    try
    {
        Console.WriteLine("Desired property change:");

        Console.WriteLine(JsonConvert.SerializeObject(desiredProperties));

        if (desiredProperties["TemperatureThreshold"]!=null)
            temperatureThreshold =
desiredProperties["TemperatureThreshold"];

    }
    catch (AggregateException ex)
    {
        foreach (Exception exception in ex.InnerExceptions)
        {
            Console.WriteLine();
            Console.WriteLine("Error when receiving desired
property: {0}", exception);
        }
    }
    catch (Exception ex)
    {
        Console.WriteLine();
        Console.WriteLine("Error when receiving desired property:
{0}", ex.Message);
    }
    return Task.CompletedTask;
}

```

7. Add the **FilterMessages** method. This method is called whenever the module receives a message from the IoT Edge hub. It filters out messages that report temperatures below the temperature threshold set via the module twin. It also adds the **MessageType** property to the message with the value set to **Alert**.

C#

```
async Task<MessageResponse> FilterMessages(Message message, object userContext)
{
    var counterValue = Interlocked.Increment(ref _counter);
    try
    {
        ModuleClient moduleClient = (ModuleClient)userContext;
        var messageBytes = message.GetBytes();
        var messageString = Encoding.UTF8.GetString(messageBytes);
        Console.WriteLine($"Received message {counterValue}:
[{messageString}]");

        // Get the message body.
        var messageBody =
JsonConvert.DeserializeObject<MessageBody>(messageString);

        if (messageBody != null && messageBody.machine.temperature > temperatureThreshold)
        {
            Console.WriteLine($"Machine temperature
{messageBody.machine.temperature} +
$"exceeds threshold {temperatureThreshold}");
            using (var filteredMessage = new Message(messageBytes))
            {
                foreach (KeyValuePair<string, string> prop in message.Properties)
                {
                    filteredMessage.Properties.Add(prop.Key,
prop.Value);
                }

                filteredMessage.Properties.Add("MessageType",
"Alert");
                await moduleClient.SendEventAsync("output1",
filteredMessage);
            }
        }

        // Indicate that the message treatment is completed.
        return MessageResponse.Completed;
    }
    catch (AggregateException ex)
    {
        foreach (Exception exception in ex.InnerExceptions)
        {
            Console.WriteLine();
        }
    }
}
```

```

        Console.WriteLine("Error in sample: {0}", exception);
    }
    // Indicate that the message treatment is not completed.
    var moduleClient = (ModuleClient)userContext;
    return MessageResponse.Abandoned;
}
catch (Exception ex)
{
    Console.WriteLine();
    Console.WriteLine("Error in sample: {0}", ex.Message);
    // Indicate that the message treatment is not completed.
    ModuleClient moduleClient = (ModuleClient)userContext;
    return MessageResponse.Abandoned;
}
}
}

```

8. Save the **ModuleBackgroundService.cs** file.
9. In the Visual Studio Code explorer, open the **deployment.template.json** file in your IoT Edge solution workspace.
10. Since we changed the name of the endpoint that the module listens on, we also need to update the routes in the deployment manifest so that the **edgeHub** sends messages to the new endpoint.

Find the **routes** section in the **\$edgeHub** module twin. Update the **sensorTofiltermodule** route to replace `input1` with `inputFromSensor`:

JSON

```

"sensorTofiltermodule": "FROM
/messages/modules/tempSensor/outputs/temperatureOutput INTO
BrokeredEndpoint(\"/modules/filtermodule/inputs/inputFromSensor\")"

```

11. Add the **filtermodule** module twin to the deployment manifest. Insert the following JSON content at the bottom of the **modulesContent** section, after the **\$edgeHub** module twin:

JSON

```

"filtermodule": {
    "properties.desired":{
        "TemperatureThreshold":25
    }
}

```

12. Save the **deployment.template.json** file.

Build and push your solution

You updated the module code and the deployment template to help understand some key deployment concepts. Now, you're ready to build your module container image and push it to your container registry.

In Visual Studio Code, open the `deployment.template.json` deployment manifest file. The [deployment manifest](#) describes the modules to be configured on the targeted IoT Edge device. Before deployment, you need to update your Azure Container Registry credentials and your module images with the proper `createOptions` values. For more information about `createOption` values, see [How to configure container create options for IoT Edge modules](#).

If you're using an Azure Container Registry to store your module image, add your credentials to the `modulesContent > edgeAgent > settings > registryCredentials` section in `deployment.template.json`. Replace `myacr` with your own registry name and provide your password and `Login server` address. For example:

JSON

```
"registryCredentials": {  
    "myacr": {  
        "username": "myacr",  
        "password": "<your_acr_password>",  
        "address": "myacr.azurecr.io"  
    }  
}
```

Add or replace the following stringified content to the `createOptions` value for each system (`edgeHub` and `edgeAgent`) and custom module (`filtermodule` and `tempSensor`) listed. Change the values if necessary.

JSON

```
"createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\": [{\"HostPort\":\"5671\"}], \"8883/tcp\": [{\"HostPort\":\"8883\"}], \"443/tcp\": [{\"HostPort\":\"443\"}]}}}
```

For example, the `filtermodule` configuration should be similar to:

JSON

```
"filtermodule": {  
    "version": "1.0",  
    "type": "docker",  
    "status": "running",  
    "restartPolicy": "always",  
    "settings": {  
        "image": "myacr.azurecr.io/filtermodule:0.0.1-amd64",  
        "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}\"}  
    }  
}
```

Build module Docker image

Open the Visual Studio Code integrated terminal by selecting **Terminal > New Terminal**.

C#

Use the `dotnet publish` command to build the container image for Linux and amd64 architecture. Change directory to the *filtermodule* directory in your project and run the *dotnet publish* command.

Bash

```
dotnet publish --os linux --arch x64 /t:PublishContainer
```

Currently, the *iotedgedev* tool template targets .NET 7.0. If you want to target a different version of .NET, you can edit the *filtermodule.csproj* file and change the *TargetFramework* and *PackageReference* values. For example to target .NET 8.0, your *filtermodule.csproj* file should look like this:

XML

```
<Project Sdk="Microsoft.NET.Sdk.Worker">  
  <PropertyGroup>  
    <TargetFramework>net8.0</TargetFramework>  
    <Nullable>enable</Nullable>  
    <ImplicitUsings>enable</ImplicitUsings>  
  </PropertyGroup>  
  <ItemGroup>  
    <PackageReference Include="Microsoft.Azure.Devices.Client"  
      Version="1.42.0" />  
    <PackageReference Include="Microsoft.Extensions.Hosting"  
      Version="8.0.0" />  
  </ItemGroup>
```

```
</ItemGroup>  
</Project>
```

Tag the docker image with your container registry information, version, and architecture. Replace **myacr** with your own registry name.

Bash

```
docker tag filtermodule myacr.azurecr.io/filtermodule:0.0.1-amd64
```

Push module Docker image

Provide your container registry credentials to Docker so that it can push your container image to storage in the registry.

1. Sign in to Docker with the Azure Container Registry (ACR) credentials.

Bash

```
docker login -u <ACR username> -p <ACR password> <ACR login server>
```

You might receive a security warning recommending the use of `--password-stdin`.

While that's a recommended best practice for production scenarios, it's outside the scope of this tutorial. For more information, see the [docker login](#) reference.

2. Sign in to the Azure Container Registry. You need to [Install Azure CLI](#) to use the `az` command. This command asks for your user name and password found in your container registry in [Settings > Access keys](#).

Azure CLI

```
az acr login -n <ACR registry name>
```

 Tip

If you get logged out at any point in this tutorial, repeat the Docker and Azure Container Registry sign in steps to continue.

3. [Push](#) your module image to the local registry or a container registry.

Bash

```
docker push <ImageName>
```

For example:

Bash

```
# Push the Docker image to the local registry

docker push localhost:5000/filtermodule:0.0.1-amd64

# Or push the Docker image to an Azure Container Registry. Replace
myacr with your Azure Container Registry name.

az acr login --name myacr
docker push myacr.azurecr.io/filtermodule:0.0.1-amd64
```

Update the deployment template

Update the deployment template *deployment.template.json* with the container registry image location. For example, if you're using an Azure Container Registry *myacr.azurecr.io* and your image is *filtermodule:0.0.1-amd64*, update the *filtermodule* configuration to:

JSON

```
"filtermodule": {
    "version": "1.0",
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
        "image": "myacr.azurecr.io/filtermodule:0.0.1-amd64",
        "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}\"}"
    }
}
```

Optional: Update the module and image

If you make changes to your module code, you need to rebuild and push the module image to your container registry. Use the steps in this section to update the build and container image. You can skip this section if you didn't make any changes to your module code.

1. Open the **module.json** file in the *filtermodule* folder.

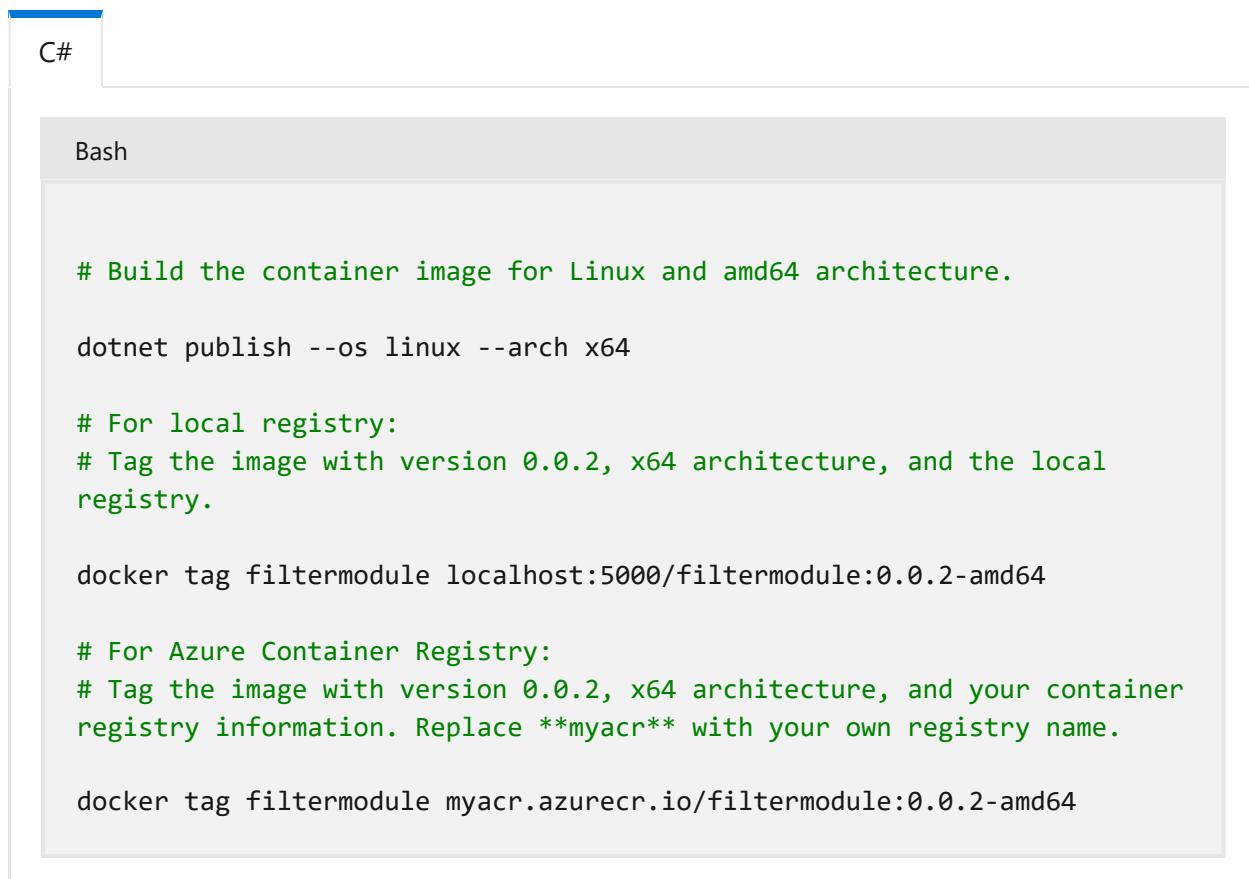
2. Change the version number for the module image. For example, increment the patch version number to "version": "0.0.2" as if you made a small fix in the module code.

 **Tip**

Module versions enable version control, and allow you to test changes on a small set of devices before deploying updates to production. If you don't increment the module version before building and pushing, then you overwrite the repository in your container registry.

3. Save your changes to the **module.json** file.

Build and push the updated image with a *0.0.2* version tag. For example, to build and push the image for the local registry or an Azure container registry, use the following commands:



The screenshot shows a terminal window with two tabs: "C#" and "Bash". The "Bash" tab is active and contains the following command-line session:

```
# Build the container image for Linux and amd64 architecture.  
dotnet publish --os linux --arch x64  
  
# For local registry:  
# Tag the image with version 0.0.2, x64 architecture, and the local  
registry.  
  
docker tag filtermodule localhost:5000/filtermodule:0.0.2-amd64  
  
# For Azure Container Registry:  
# Tag the image with version 0.0.2, x64 architecture, and your container  
registry information. Replace **myacr** with your own registry name.  
  
docker tag filtermodule myacr.azurecr.io/filtermodule:0.0.2-amd64
```

Open the **deployment.amd64.json** file again. Notice the build system doesn't create a new file when you run the build and push command again. Rather, the same file updates to reflect the changes. The *filtermodule* image now points to the 0.0.2 version of the container.

To further verify what the build and push command did, go to the [Azure portal](#) and navigate to your container registry.

In your container registry, select **Repositories** then **filtermodule**. Verify that both versions of the image push to the registry.

The screenshot shows the Azure Container Registry interface. On the left, there's a sidebar with 'myregistry | Repositories' selected under 'Services'. The main area shows a list of repositories: 'azuriedgetedge-agent', 'cmodule', 'est', 'filtermodule', and 'samplemodule'. The 'filtermodule' repository is highlighted with a red box. On the right, the details for the 'samplemodule' repository are shown. It has a tag count of 2 and a manifest count of 2. The tags listed are '0.0.1-amd64' and '0.0.2-amd64', both of which are also highlighted with a red box. The 'Tags' column lists the tag names, the 'Digest' column lists their respective Docker image digests, and the 'Last modified' column shows the timestamp for each tag.

Tags ↑↓	Digest ↑↓	Last modified
0.0.1-amd64	sha256:3db0feedb14acac8c4d24dc...	11/8/2022, 2:24 PM PST
0.0.2-amd64	sha256:3db0feedb14acac8c4d24dc...	11/8/2022, 3:20 PM PST

Troubleshoot

If you encounter errors when building and pushing your module image, it often has to do with Docker configuration on your development machine. Use the following checks to review your configuration:

- Did you run the `docker login` command using the credentials that you copied from your container registry? These credentials are different than the ones that you use to sign in to Azure.
- Is your container repository correct? Does it have your correct container registry name and your correct module name? Open the `module.json` file in the `filtermodule` folder to check. The repository value should look like `<registry name>.azurecr.io/filtermodule`.
- If you used a different name than `filtermodule` for your module, is that name consistent throughout the solution?
- Is your machine running the same type of containers that you're building? This tutorial is for Linux IoT Edge devices, so Visual Studio Code should say `amd64` or `arm32v7` in the side bar, and Docker Desktop should be running Linux containers.

Deploy modules to device

You verified that there are built container images stored in your container registry, so it's time to deploy them to a device. Make sure that your IoT Edge device is up and running.

Use the [IoT Edge Azure CLI set-modules](#) command to deploy the modules to the Azure IoT Hub. For example, to deploy the modules defined in the `deployment.template.json`

file to IoT Hub *my-iot-hub* for the IoT Edge device *my-device*, use the following command. Replace the values for **hub-name**, **device-id**, and **login** IoT Hub connection string with your own.

Azure CLI

```
az iot edge set-modules --hub-name my-iot-hub --device-id my-device --content ./deployment.template.json --login "HostName=my-iot-hub.azure-devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=<SharedAccessKey>"
```

Tip

You can find your IoT Hub connection string including the shared access key in the Azure portal. Go to your IoT Hub > **Security settings** > **Shared access policies** > **iothubowner**.

View changes on device

If you want to see what's happening on your device itself, use the commands in this section to inspect the IoT Edge runtime and modules running on your device.

The commands in this section are for your IoT Edge device, not your development machine. If you're using a virtual machine for your IoT Edge device, connect to it now. In Azure, go to the virtual machine's overview page and select **Connect** to access the secure shell connection.

- View all modules deployed to your device, and check their status:

Bash

```
iotedge list
```

You should see four modules: the two IoT Edge runtime modules, *tempSensor*, and *filtermodule*. You should see all four listed as running.

- Inspect the logs for a specific module:

Bash

```
iotedge logs <module name>
```

IoT Edge modules are case-sensitive.

The *tempSensor* and *filtermodule* logs should show the messages they're processing. The *edgeAgent* module is responsible for starting the other modules, so its logs have information about implementing the deployment manifest. If you find a module is unlisted or not running, the *edgeAgent* logs likely have the errors. The *edgeHub* module is responsible for communications between the modules and IoT Hub. If the modules are up and running, but the messages aren't arriving at your IoT hub, the *edgeHub* logs likely have the errors.

Clean up resources

If you plan to continue to the next recommended article, you can keep the resources and configurations that you created and reuse them. You can also keep using the same IoT Edge device as a test device.

Otherwise, you can delete the local configurations and the Azure resources that you used in this article to avoid charges.

Delete Azure resources

Deleting Azure resources and resource groups is irreversible. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the IoT hub inside an existing resource group that has resources that you want to keep, delete only the IoT hub resource itself, not the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources that are contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can click into each resource to delete them individually.

Next steps

In this tutorial, you set up Visual Studio Code on your development machine and deployed your first IoT Edge module that contains code to filter raw data generated by your IoT Edge device.

You can continue on to the next tutorials to learn how Azure IoT Edge can help you deploy Azure cloud services to process and analyze data at the edge.

[Debug Azure IoT Edge modules](#)

[Functions](#)

[Stream Analytics](#)

[Custom Vision Service](#)

Use Visual Studio 2022 to develop and debug modules for Azure IoT Edge

Article • 07/17/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article shows you how to use Visual Studio 2022 to develop, debug, and deploy custom Azure IoT Edge modules. Visual Studio 2022 provides templates for IoT Edge modules written in C and C#. The supported device architectures are Windows x64, Linux x64, ARM32, and ARM64 (preview). For more information about supported operating systems, languages, and architectures, see [Language and architecture support](#).

This article includes steps for two IoT Edge development tools.

- Command line interface (CLI) is the preferred tool for development.
- **Azure IoT Edge tools for Visual Studio** extension. The extension is in [maintenance mode](#).

Use the tool selector button at the beginning to choose your tool option for this article. Both tools provide the following benefits:

- Create, edit, build, run, and debug IoT Edge solutions and modules on your local development computer.
- Code your Azure IoT modules in C or C# with the benefits of Visual Studio development.
- Deploy your IoT Edge solution to an IoT Edge device via Azure IoT Hub.

Prerequisites

This article assumes that you use a machine running Windows as your development machine.

- Install or modify Visual Studio 2022 on your development machine. Choose the **Azure development** and **Desktop development with C++** workloads options.
- Download and install [Azure IoT Edge Tools](#) from the Visual Studio Marketplace. You can use the Azure IoT Edge Tools extension to create and build your IoT Edge solution. The preferred development tool is the command-line (CLI) *Azure IoT Edge Dev Tool*. The extension includes the Azure IoT Edge project templates used to create the Visual Studio project. Currently, you need the extension installed regardless of the development tool you use.

 **Important**

The *Azure IoT Edge Tools for VS 2022* extension is in [maintenance mode](#).

The preferred development tool is the command-line (CLI) *Azure IoT Edge Dev Tool*.

 **Tip**

If you are using Visual Studio 2019, download and install [Azure IoT Edge Tools for VS 2019](#) from the Visual Studio marketplace.

- Install the **Vcpkg** library manager

Windows Command Prompt

```
git clone https://github.com/Microsoft/vcpkg  
cd vcpkg  
bootstrap-vcpkg.bat
```

Install the **azure-iot-sdk-c** package for Windows

Windows Command Prompt

```
vcpkg.exe install azure-iot-sdks:x64-windows  
vcpkg.exe --triplet x64-windows integrate install
```

- Download and install a [Docker compatible container management system](#) on your development machine to build and run your module images. For example, install [Docker Community Edition](#).
- To develop modules with **Linux containers**, use a Windows computer that meets the [requirements for Docker Desktop](#).

- Create an [Azure Container Registry](#) or [Docker Hub](#) to store your module images.

💡 Tip

You can use a local Docker registry for prototype and testing purposes instead of a cloud registry.

- Install the [Azure CLI](#).
- To test your module on a device, you need an active IoT Hub with at least one IoT Edge device. To create an IoT Edge device for testing, you can create one in the Azure portal or with the CLI:
 - Creating one in the [Azure portal](#) is the quickest. From the Azure portal, go to your IoT Hub resource. Select **Devices** under the **Device management** menu and then select **Add Device**.

In **Create a device**, name your device using **Device ID**, check **IoT Edge Device**, then select **Save** in the lower left.

Finally, confirm that your new device exists in your IoT Hub, from the **Device management > Devices** menu. For more information on creating an IoT Edge device through the Azure portal, read [Create and provision an IoT Edge device on Linux using symmetric keys](#).

- To create an IoT Edge device with the CLI, follow the steps in the quickstart for [Linux](#) or [Windows](#). In the process of registering an IoT Edge device, you create an IoT Edge device.

If you're running the IoT Edge daemon on your development machine, you might need to stop EdgeHub and EdgeAgent before you start development in Visual Studio.

Create an Azure IoT Edge project

The IoT Edge project template in Visual Studio creates a solution to deploy to IoT Edge devices. First, you create an Azure IoT Edge solution. Then, you create a module in that solution. Each IoT Edge solution can contain more than one module.

⚠️ Warning

The Azure IoT Edge tools for Visual Studio extension is missing the project templates for C and C# modules. We are working to resolve the issue. If you can't create IoT Edge modules using the extension, use the following workaround.

Download the following files and place them in the listed Visual Studio template directory:

[\[+\] Expand table](#)

Template file	Add to directory
azureiotedgeModule-v0.0.4.zip	%userprofile%\Documents\Visual Studio 2022\Templates\ProjectTemplates\Visual C#
azureiotedgeVCModulevs17-v0.0.9.zip	%userprofile%\Documents\Visual Studio 2022\Templates\ProjectTemplates\Visual C++ Project

In our solution, we're going to build three projects. The main module that contains *EdgeAgent* and *EdgeHub*, in addition to the temperature sensor module. Next, you add two more IoT Edge modules.

Important

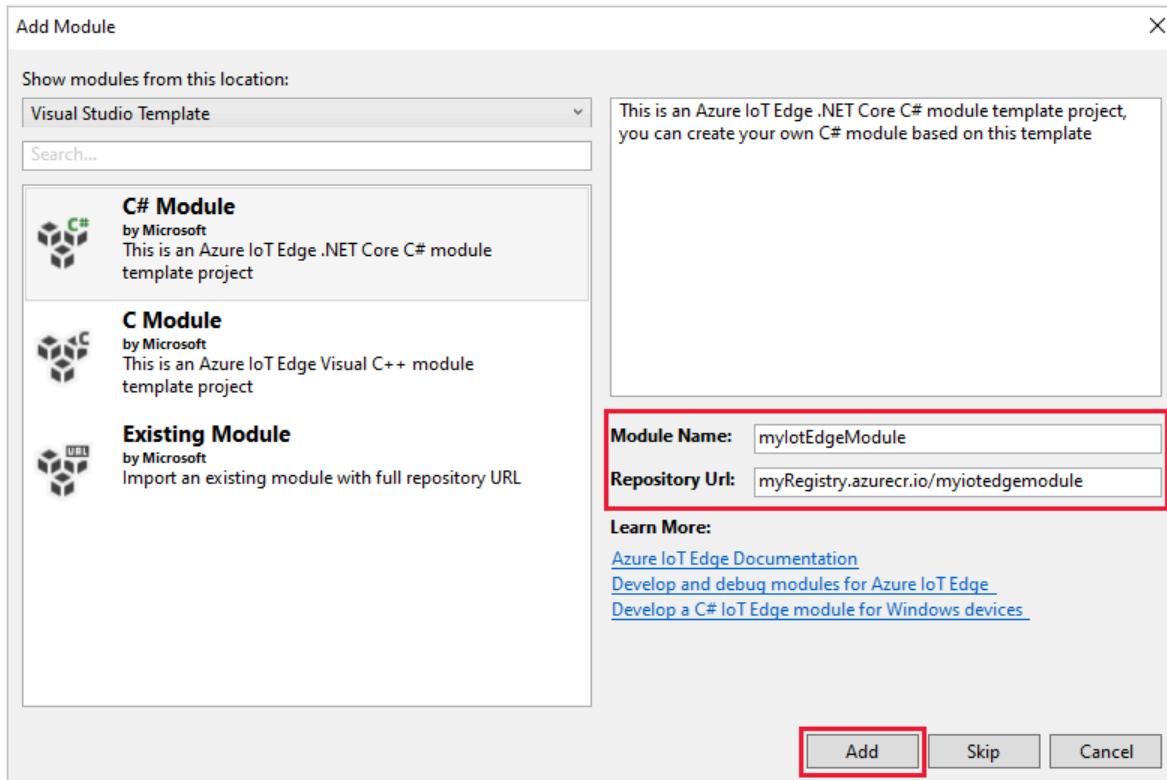
The IoT Edge project structure created by Visual Studio isn't the same as the one in Visual Studio Code.

Currently, the Azure IoT Edge Dev Tool CLI doesn't support creating the Visual Studio project type. You need to use the Visual Studio IoT Edge extension to create the Visual Studio project.

1. In Visual Studio, create a new project.
2. In **Create a new project**, search for **Azure IoT Edge**. Select the project that matches the platform and architecture for your IoT Edge device, and select **Next**.
3. In **Configure your new project**, enter a name for your project, specify the location, and select **Create**.
4. In **Add Module**, select the type of module you want to develop. If you have an existing module you want to add to your deployment, select **Existing module**.
5. In **Module Name**, enter a name for your module. Choose a name that's unique within your container registry.

6. In **Repository Url**, provide the name of the module's image repository. Visual Studio autopopulates the module name with **localhost:5000/<your module name>**. Replace it with your own registry information. Use **localhost** if you use a local Docker registry for testing. If you use Azure Container Registry, then use the login server from your registry's settings. The login server looks like **<registry name>.azurecr.io**. Only replace the **localhost:5000** part of the string so that the final result looks like **<registry name>.azurecr.io/<your module name>**.

7. Select **Add** to add your module to the project.



ⓘ Note

If you have an existing IoT Edge project, you can change the repository URL by opening the **module.json** file. The repository URL is located in the **repository** property of the JSON file.

Now, you have an IoT Edge project and an IoT Edge module in your Visual Studio solution.

Project structure

In your solution, there are two project level folders including a main project folder and a single module folder. For example, you may have a main project folder named

AzureIoTEdgeApp1 and a module folder named *IoTEdgeModule1*. The main project folder contains your deployment manifest.

The module project folder contains a file for your module code named either `Program.cs` or `main.c` depending on the language you chose. This folder also contains a file named `module.json` that describes the metadata of your module. Various Docker files included here provide the information needed to build your module as a Windows or Linux container.

Deployment manifest of your project

The deployment manifest you edit is named `deployment.debug.template.json`. This file is a template of an IoT Edge deployment manifest that defines all the modules that run on a device along with how they communicate with each other. For more information about deployment manifests, see [Learn how to deploy modules and establish routes](#).

If you open this deployment template, you see that the two runtime modules, `edgeAgent` and `edgeHub` are included, along with the custom module that you created in this Visual Studio project. A fourth module named `SimulatedTemperatureSensor` is also included. This default module generates simulated data that you can use to test your modules, or delete if it's not necessary. To see how the simulated temperature sensor works, view the [SimulatedTemperatureSensor.csproj source code](#).

Set IoT Edge runtime version

Currently, the latest stable runtime version is 1.5. You should update the IoT Edge runtime version to the latest stable release or the version you want to target for your devices.

1. Open `deployment.debug.template.json` deployment manifest file. The [deployment manifest](#) is a JSON document that describes the modules to be configured on the targeted IoT Edge device.
2. Change the runtime version for the system runtime module images `edgeAgent` and `edgeHub`. For example, if you want to use the IoT Edge runtime version 1.5, change the following lines in the deployment manifest file:

JSON

```
"systemModules": {  
    "edgeAgent": {  
        //...  
        "image": "mcr.microsoft.com/azureiotedge-agent:1.5",  
    }  
}
```

```
//...
"edgeHub": {
//...
    "image": "mcr.microsoft.com/azureiotedge-hub:1.5",
//...
```

Module infrastructure & development options

When you add a new module, it comes with default code that is ready to be built and deployed to a device so that you can start testing without touching any code. The module code is located within the module folder in a file named `Program.cs` (for C#) or `main.c` (for C).

The default solution is built so that the simulated data from the **SimulatedTemperatureSensor** module is routed to your module, which takes the input and then sends it to IoT Hub.

When you're ready to customize the module template with your own code, use the [Azure IoT Hub SDKs](#) to build modules that address the key needs for IoT solutions such as security, device management, and reliability.

Build module Docker image

Once you've developed your module, you can build the module image to store in a container registry for deployment to your IoT Edge device.

Use the module's Dockerfile to build the module Docker image.

Bash

```
docker build --rm -f "<DockerFilePath>" -t <ImageNameAndTag> "<ContextPath>"
```

For example, let's assume your command shell is in your project directory and your module name is *IotEdgeModule1*. To build the image for the local registry or an Azure container registry, use the following commands:

Bash

```
# Build the image for the local registry

docker build --rm -f "./IotEdgeModule1/Dockerfile.amd64.debug" -t
localhost:5000/iotedgemodule1:0.0.1-amd64 "./IotEdgeModule1"

# Or build the image for an Azure Container Registry
```

```
docker build --rm -f "./IotEdgeModule1/Dockerfile.amd64.debug" -t myacr.azurecr.io/iotedgemodule1:0.0.1-amd64 "./IotEdgeModule1"
```

Push module Docker image

Push your module image to the local registry or a container registry.

```
docker push <ImageName>
```

For example:

Bash

```
# Push the Docker image to the local registry

docker push localhost:5000/iotedgemodule1:0.0.1-amd64

# Or push the Docker image to an Azure Container Registry
az acr login --name myacr
docker push myacr.azurecr.io/iotedgemodule1:0.0.1-amd64
```

Deploy the module to the IoT Edge device.

In Visual Studio, open `deployment.debug.template.json` deployment manifest file in the main project. The [deployment manifest](#) is a JSON document that describes the modules to be configured on the targeted IoT Edge device. Before deployment, you need to update your Azure Container Registry credentials, your module images, and the proper `createOptions` values. For more information about `createOption` values, see [How to configure container create options for IoT Edge modules](#).

1. If you're using an Azure Container Registry to store your module image, you need to add your credentials to `deployment.debug.template.json` in the `edgeAgent` settings. For example,

JSON

```
"modulesContent": {
"$edgeAgent": {
"properties.desired": {
"schemaVersion": "1.1",
"runtime": {
"type": "docker",
"settings": {
"minDockerVersion": "v1.25",
```

```
        "loggingOptions": "",  
        "registryCredentials": {  
            "myacr": {  
                "username": "myacr",  
                "password": "<your_acr_password>",  
                "address": "myacr.azurecr.io"  
            }  
        }  
    },  
    //...
```

2. Replace the *image* property value with the module image name you pushed to the registry. For example, if you pushed an image tagged

`myacr.azurecr.io/iotedgemodule1:0.0.1-amd64` for custom module *iotEdgeModule1*, replace the image property value with the tag value.

3. Add or replace the *createOptions* value with stringified content *for each system and custom module in the deployment template*.

For example, the *iotEdgeModule1*'s *image* and *createOptions* settings would be similar to the following:

JSON

```
"IoTEdgeModule1": {  
    "version": "1.0.0",  
    "type": "docker",  
    "status": "running",  
    "restartPolicy": "always",  
    "settings": {  
        "image": "myacr.azurecr.io/iotedgemodule1:0.0.1-amd64",  
        "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}\"}  
    }  
}
```

Use the [IoT Edge Azure CLI set-modules](#) command to deploy the modules to the Azure IoT Hub. For example, to deploy the modules defined in the *deployment.debug.amd64.json* file to IoT Hub *my-iot-hub* for the IoT Edge device *my-device*, use the following command:

Azure CLI

```
az iot edge set-modules --hub-name my-iot-hub --device-id my-device --  
content ./deployment.debug.template.json --login "HostName=my-iot-hub.azure-
```

```
devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=
<SharedAccessKey>"
```

💡 Tip

You can find your IoT Hub connection string in the Azure portal under Azure IoT Hub > Security settings > Shared access policies.

Confirm the deployment to your device

To check that your IoT Edge modules were deployed to Azure, sign in to your device (or virtual machine), for example through SSH or Azure Bastion, and run the IoT Edge list command.

Azure CLI

```
iotedge list
```

You should see a list of your modules running on your device or virtual machine.

Azure CLI

NAME	STATUS	DESCRIPTION	CONFIG
SimulatedTemperatureSensor	running	Up a minute	
mcr.microsoft.com.azureiotedge-simulated-temperature-sensor:1.0			
edgeAgent	running	Up a minute	
mcr.microsoft.com.azureiotedge-agent:1.2			
edgeHub	running	Up a minute	
mcr.microsoft.com.azureiotedge-hub:1.2			
IotEdgeModule1	running	Up a minute	
myacr.azurecr.io/iotedgemodule1:0.0.1-amd64.debug			
myIotEdgeModule2	running	Up a minute	
myacr.azurecr.io/myiotedgemodule2:0.0.1-amd64.debug			

Debug using Docker Remote SSH

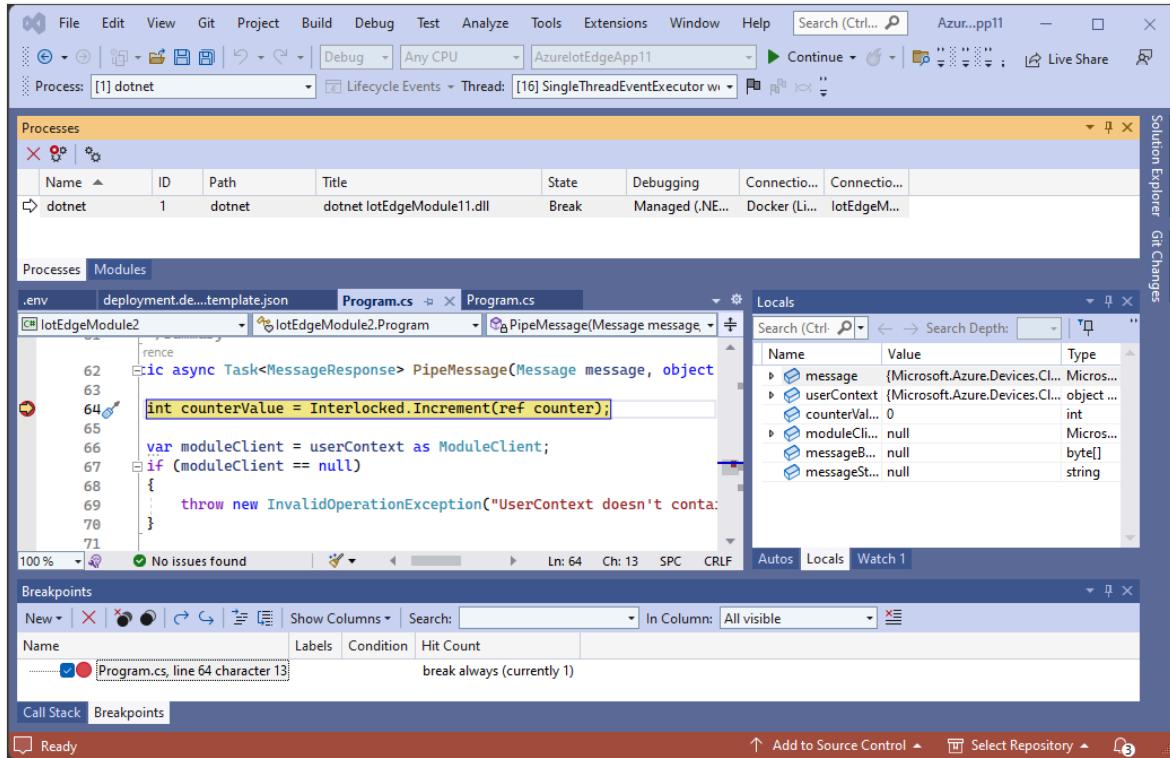
The Docker and Moby engines support SSH connections to containers allowing you to attach and debug code on a remote device using Visual Studio.

1. Connecting remotely to Docker requires root-level privileges. Follow the steps in [Manage docker as a non-root user](#) to allow connection to the Docker daemon on the remote device. When you're finished debugging, you may want to remove your user from the Docker group.

2. Follow the steps to use Visual Studio to [Attach to a process running on a Docker container](#) on your remote device.

3. In Visual Studio, set breakpoints in your custom module.

4. When a breakpoint is hit, you can inspect variables, step through code, and debug your module.



Next steps

- To develop custom modules for your IoT Edge devices, [Understand and use Azure IoT Hub SDKs](#).
- To monitor the device-to-cloud (D2C) messages for a specific IoT Edge device, review the [Tutorial: Monitor IoT Edge devices](#) to get started.

Feedback

Was this page helpful?

Yes

No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Tutorial: Deploy Azure Functions as IoT Edge modules

Article • 06/10/2024

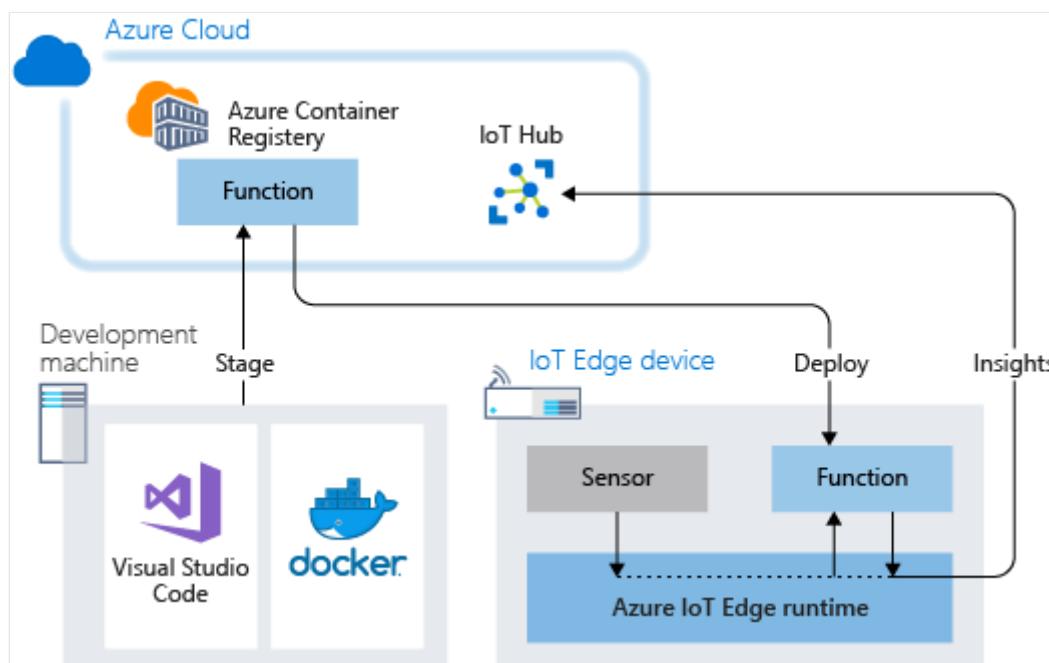
Applies to: ✓ IoT Edge 1.5 ✓ IoT Edge 1.4

ⓘ Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

You can use Azure Functions to deploy code that implements your business logic directly to your Azure IoT Edge devices. This tutorial walks you through creating and deploying an Azure Function that filters sensor data on the simulated IoT Edge device. You use the simulated IoT Edge device that you created in the quickstarts. In this tutorial, you learn how to:

- ✓ Use Visual Studio Code to create an Azure Function.
- ✓ Use Visual Studio Code and Docker to create a Docker image and publish it to a container registry.
- ✓ Deploy the module from the container registry to your IoT Edge device.
- ✓ View filtered data.



The Azure Function that you create in this tutorial filters the temperature data that's generated by your device. The Function only sends messages upstream to Azure IoT

Hub when the temperature is above a specified threshold.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

Before beginning this tutorial, do the tutorial to set up your development environment for Linux container development: [Develop Azure IoT Edge modules using Visual Studio Code](#). After completing that tutorial, you should have the following prerequisites in place:

- A free or standard-tier [IoT Hub](#) in Azure.
- An AMD64 device running Azure IoT Edge with Linux containers. You can use the quickstart to set up a [Linux device](#) or [Windows device](#).
- A container registry, like [Azure Container Registry](#).
- [Visual Studio Code](#) configured with the [Azure IoT Edge](#) and [Azure IoT Hub](#) extensions. The *Azure IoT Edge tools for Visual Studio Code* extension is in [maintenance mode](#).
- Download and install a [Docker compatible container management system](#) on your development machine. Configure it to run Linux containers.

To develop an IoT Edge module with Azure Functions, install additional prerequisites on your development machine:

- [C# for Visual Studio Code \(powered by OmniSharp\) extension](#).
- [The .NET Core SDK](#).

Create a function project

The Azure IoT Edge for Visual Studio Code that you installed in the prerequisites provides management capabilities and some code templates. In this section, you use Visual Studio Code to create an IoT Edge solution that contains an Azure Function.

Create a new project

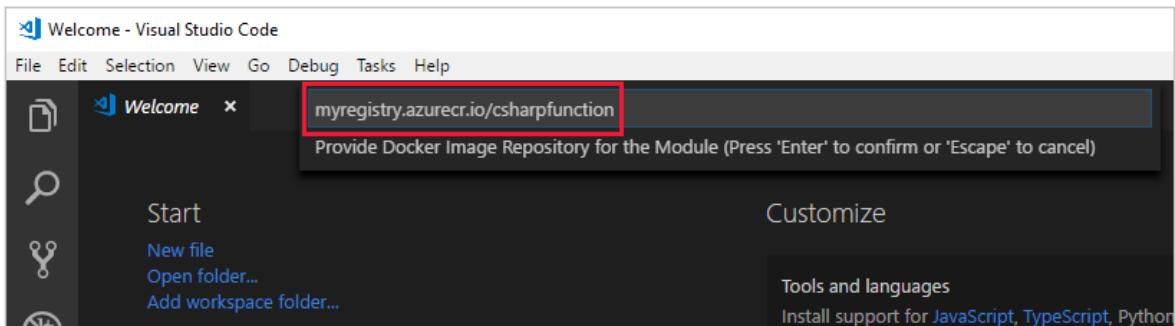
Follow these steps to create a C# Function solution template that's customizable.

1. Open Visual Studio Code on your development machine.
2. Open the Visual Studio Code command palette by selecting **View > Command Palette**.

3. In the command palette, add and run the command **Azure IoT Edge: New IoT**

Edge solution. Follow these prompts in the command palette to create your solution:

- Select a folder: choose the location on your development machine for Visual Studio Code to create the solution files.
- Provide a solution name: add a descriptive name for your solution, like **FunctionSolution**, or accept the default.|
- Select a module template: choose **Azure Functions - C#**.
- Provide a module name | Name your module **CSharpFunction**.
- Provide a Docker image repository for the module. An image repository includes the name of your container registry and the name of your container image. Your container image is pre-populated from the last step. Replace **localhost:5000** with the **Login server** value from your Azure container registry. You can retrieve the **Login server** from the **Overview** page of your container registry in the Azure portal. The final string looks like <registry name>.azurecr.io/csharpfunction.



Add your registry credentials

The environment file of your solution stores the credentials for your container registry and shares them with the IoT Edge runtime. The runtime needs these credentials to pull your private images onto your IoT Edge device.

The IoT Edge extension in Visual Studio Code tries to pull your container registry credentials from Azure and populate them in the environment file. Check to see if your credentials are already in the file. If not, add them now:

1. In the Visual Studio Code explorer, open the `.env` file.
2. Update the fields with the **username** and **password** values that you copied from your Azure container registry. You can find them again by going to your container registry in Azure and looking on the **Settings > Access keys** page.
3. Save this file.

(!) Note

This tutorial uses admin login credentials for Azure Container Registry, which are convenient for development and test scenarios. When you're ready for production scenarios, we recommend a least-privilege authentication option like service principals. For more information, see [Manage access to your container registry](#).

Set target architecture to AMD64

Running Azure Functions modules on IoT Edge is supported only on Linux AMD64 based containers. The default target architecture for Visual Studio Code is Linux AMD64, but we set it explicitly to Linux AMD64 here.

1. Open the command palette and search for **Azure IoT Edge: Set Default Target Platform for Edge Solution**.
2. In the command palette, select the AMD64 target architecture from the list of options.

Update the module with custom code

Let's add some additional code so your **CSharpFunction** module processes the messages at the edge before forwarding them to IoT Hub.

1. In the Visual Studio Code explorer, open **modules > CSharpFunction > CSharpFunction.cs**.
2. Replace the contents of the **CSharpFunction.cs** file with the following code. This code receives telemetry about ambient and machine temperature, and only forwards the message on to IoT Hub if the machine temperature is above a defined threshold.

C#

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Azure.Devices.Client;
using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.EdgeHub;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
```

```

using Newtonsoft.Json;

namespace Functions.Samples
{
    public static class CSharpFunction
    {
        [FunctionName("CSharpFunction")]
        public static async Task FilterMessageAndSendMessage(
            [EdgeHubTrigger("input1")] Message messageReceived,
            [EdgeHub(OutputName = "output1")] IAsyncCollector<Message>
output,
            ILogger logger)
        {
            const int temperatureThreshold = 20;
            byte[] messageBytes = messageReceived.GetBytes();
            var messageString =
                System.Text.Encoding.UTF8.GetString(messageBytes);

            if (!string.IsNullOrEmpty(messageString))
            {
                logger.LogInformation("Info: Received one non-empty
message");
                // Get the body of the message and deserialize it.
                var messageBody =
                    JsonConvert.DeserializeObject<MessageBody>(messageString);

                if (messageBody != null &&
messageBody.machine.temperature > temperatureThreshold)
                {
                    // Send the message to the output as the
temperature value is greater than the threshold.
                    using (var filteredMessage = new
Message(messageBytes))
                    {
                        // Copy the properties of the original message
into the new Message object.
                        foreach (KeyValuePair<string, string> prop in
messageReceived.Properties)
                            {filteredMessage.Properties.Add(prop.Key,
prop.Value);}
                        // Add a new property to the message to
indicate it is an alert.
                        filteredMessage.Properties.Add("MessageType",
"Alert");
                        // Send the message.
                        await output.AddAsync(filteredMessage);
                        logger.LogInformation("Info: Received and
transferred a message with temperature above the threshold");
                    }
                }
            }
        }
    }
    //Define the expected schema for the body of incoming messages.
    class MessageBody

```

```
{  
    public Machine machine {get; set;}  
    public Ambient ambient {get; set;}  
    public string timeCreated {get; set;}  
}  
class Machine  
{  
    public double temperature {get; set;}  
    public double pressure {get; set;}  
}  
class Ambient  
{  
    public double temperature {get; set;}  
    public int humidity {get; set;}  
}  
}
```

3. Save the file.

Build and push your IoT Edge solution

In the previous section, you created an IoT Edge solution and modified the **CSharpFunction** to filter out messages with reported machine temperatures below the acceptable threshold. Now you need to build the solution as a container image and push it to your container registry.

1. Open the Visual Studio Code integrated terminal by selecting **View > Terminal**.
2. Sign in to Docker by entering the following command in the terminal. Sign in with the username, password, and login server from your Azure container registry. You can retrieve these values from the **Access keys** section of your registry in the Azure portal.

Bash

```
docker login -u <ACR username> -p <ACR password> <ACR login server>
```

You may receive a security warning recommending the use of `--password-stdin`. While that best practice is recommended for production scenarios, it's outside the scope of this tutorial. For more information, see the [docker login](#) reference.

3. In the Visual Studio Code explorer, right-click the **deployment.template.json** file and select **Build and Push IoT Edge Solution**.

The build and push command starts three operations. First, it creates a new folder in the solution called **config** that holds the full deployment manifest, which is built

out of information in the deployment template and other solution files. Second, it runs `docker build` to build the container image based on the appropriate dockerfile for your target architecture. Then, it runs `docker push` to push the image repository to your container registry.

This process may take several minutes the first time, but is faster the next time that you run the commands.

View your container image

Visual Studio Code outputs a success message when your container image is pushed to your container registry. If you want to confirm the successful operation for yourself, you can view the image in the registry.

1. In the Azure portal, browse to your Azure container registry.
2. Select **Services > Repositories**.
3. You should see the **csharpfunction** repository in the list. Select this repository to see more details.
4. In the **Tags** section, you should see the **0.0.1-amd64** tag. This tag indicates the version and platform of the image that you built. These values are set in the `module.json` file in the `CSharpFunction` folder.

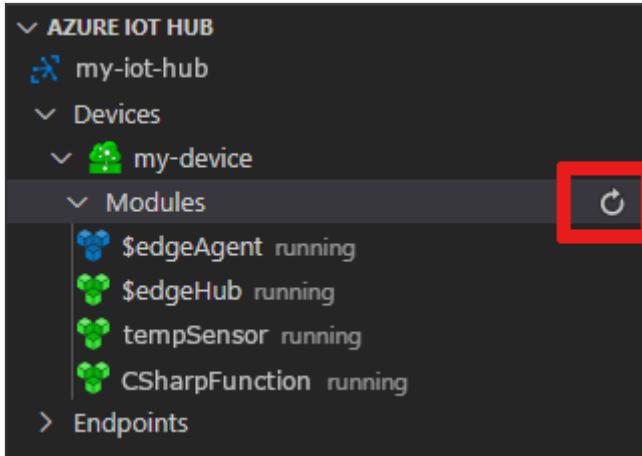
Deploy and run the solution

You can use the Azure portal to deploy your Function module to an IoT Edge device like you did in the quickstart. You can also deploy and monitor modules from within Visual Studio Code. The following sections use the Azure IoT Edge and IoT Hub for Visual Studio Code that was listed in the prerequisites. Install the extensions now, if you haven't already.

1. In the Visual Studio Code explorer, under the **Azure IoT Hub** section, expand **Devices** to see your list of IoT devices.
2. Right-click the name of your IoT Edge device, and then select **Create Deployment for Single Device**.
3. Browse to the solution folder that contains the **CSharpFunction**. Open the config folder, select the `deployment.amd64.json` file, and then choose **Select Edge Deployment Manifest**.
4. Under your device, expand **Modules** to see a list of deployed and running modules. Select the refresh button. You should see the new **CSharpFunction**

running along with the `SimulatedTemperatureSensor` module and the `$edgeAgent` and `$edgeHub`.

It may take a few moments for the new modules to show up. Your IoT Edge device has to retrieve its new deployment information from IoT Hub, start the new containers, and then report the status back to IoT Hub.



View the generated data

You can see all of the messages that arrive at your IoT hub from all your devices by running **Azure IoT Hub: Start Monitoring Built-in Event Endpoint** in the command palette. To stop monitoring messages, run the command **Azure IoT Hub: Stop Monitoring Built-in Event Endpoint** in the command palette.

You can also filter the view to see all of the messages that arrive at your IoT hub from a specific device. Right-click the device in the **Azure IoT Hub > Devices** section of the Visual Studio Code explorer and select **Start Monitoring Built-in Event Endpoint**.

Clean up resources

If you plan to continue to the next recommended article, you can keep the resources and configurations that you created and reuse them. You can also keep using the same IoT Edge device as a test device.

Otherwise, you can delete the local configurations and the Azure resources that you created in this article to avoid charges.

Delete Azure resources

Deleting Azure resources and resource groups is irreversible. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the IoT hub

inside an existing resource group that has resources that you want to keep, delete only the IoT hub resource itself, not the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources that are contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can click into each resource to delete them individually.

Next steps

In this tutorial, you created an Azure Function module with code to filter raw data that's generated by your IoT Edge device.

Continue on to the next tutorials to learn other ways that Azure IoT Edge can help you turn data into business insights at the edge.

[Find averages by using a floating window in Azure Stream Analytics](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Tutorial: Deploy Azure Stream Analytics as an IoT Edge module

Article • 04/09/2024

Applies to: ✓ IoT Edge 1.5 ✓ IoT Edge 1.4

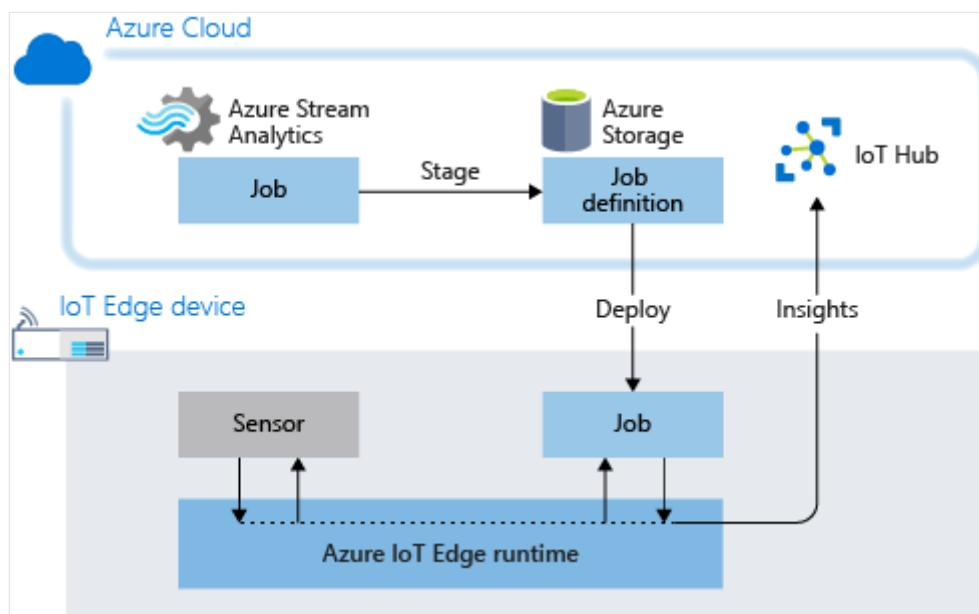
ⓘ Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

In this tutorial, you create an Azure Stream Analytics job in the Azure portal and then deploy it as an IoT Edge module with no extra code.

You learn how to:

- ✓ Create an Azure Stream Analytics job to process data on the edge.
- ✓ Connect the new Azure Stream Analytics job with other IoT Edge modules.
- ✓ Deploy the Azure Stream Analytics job to an IoT Edge device from the Azure portal.



The Stream Analytics module in this tutorial calculates the average temperature over a rolling 30-second window. When that average reaches 70, the module sends an alert for the device to take action. In this case, that action is to reset the simulated temperature sensor. In a production environment, you might use this functionality to shut off a machine or take preventative measures when the temperature reaches dangerous levels.

Why use Azure Stream Analytics in IoT Edge?

Many IoT solutions use analytics services to gain insight about data as it arrives in the cloud from IoT devices. With Azure IoT Edge, you can take [Azure Stream Analytics](#) logic and move it onto the device itself. By processing telemetry streams at the edge, you can reduce the amount of uploaded data and reduce the time it takes to react to actionable insights. Azure IoT Edge and Azure Stream Analytics are integrated to simplify your workload development.

Azure Stream Analytics provides a richly structured query syntax for data analysis, both in the cloud and on IoT Edge devices. For more information, see [Azure Stream Analytics documentation](#).

Prerequisites

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

- An Azure IoT Edge device.

You can use an Azure virtual machine as an IoT Edge device by following the steps in the quickstart for [Linux](#) or [Windows](#) devices.

- A free or standard-tier [IoT Hub](#) in Azure.

Create an Azure Stream Analytics job

In this section, you create an Azure Stream Analytics job that does the following steps:

- Receive data from your IoT Edge device.
- Query the telemetry data for values outside a set range.
- Take action on the IoT Edge device based on the query results.

Create a storage account

When you create an Azure Stream Analytics job to run on an IoT Edge device, it needs to be stored in a way that can be called from the device. You can use an existing Azure Storage account, or create a new one now.

1. In the Azure portal, go to [Create a resource > Storage > Storage account](#).
2. Provide the following values to create your storage account:

[+] Expand table

Field	Value
Subscription	Choose the same subscription as your IoT hub.
Resource group	We recommend that you use the same resource group for all of your test resources for the IoT Edge quickstarts and tutorials. For example, IoTEdgeResources .
Name	Provide a unique name for your storage account.
Location	Choose a location close to you.

3. Keep the default values for the other fields and select **Review + Create**.

4. Review your settings then select **Create**.

Create a new job

1. In the [Azure portal](#), select:

- Create a resource
- Internet of Things from the menu on the left
- Type **Stream Analytics** in the search bar to find it in the Marketplace
- Select **Create**, then **Stream Analytics job** from the dropdown menu

2. Provide the following values to create your new Stream Analytics job:

[+] Expand table

Field	Value
Name	Provide a name for your job. For example, IoTEdgeJob
Subscription	Choose the same subscription as your IoT hub.
Resource group	We recommend using the same resource group for all test resources you create during the IoT Edge quickstarts and tutorials. For example, a resource named IoTEdgeResources .
Region	Choose a location close to you.
Hosting environment	Select Edge . This option means deployment goes to an IoT Edge device instead of being hosted in the cloud.

3. Select **Review + create**.

4. Confirm your options, then select **Create**.

Configure your job

Once your Stream Analytics job is created in the Azure portal, you can configure it with an *input*, an *output*, and a *query* to run on the data that passes through.

This section creates a job that receives temperature data from an IoT Edge device. It analyzes that data in a rolling, 30-second window. If the average temperature in that window goes over 70 degrees, an alert is sent to the IoT Edge device.

ⓘ Note

You specify exactly where the data comes from and goes to in the next section, [Configure IoT Edge settings](#), when you deploy the job.

Set your input and output

1. Navigate to your Stream Analytics job in the Azure portal.
2. Under **Job topology**, select **Inputs** then **Add input**.
3. Choose **Edge Hub** from the drop-down list.

If you don't see the **Edge Hub** option in the list, then you may have created your Stream Analytics job as a cloud-hosted job. Try creating a new job and be sure to select **Edge** as the hosting environment.

4. In the **New input** pane, enter **temperature** as the **Input alias**.
5. Keep the default values for the other fields, and select **Save**.
6. Under **Job Topology**, open **Outputs** then select **Add**.
7. Choose **Edge Hub** from the drop-down list.
8. In the **New output** pane, enter **alert** as the output alias.
9. Keep the default values for the other fields, and select **Save**.

Create a query

1. Under **Job Topology**, select **Query**.
2. Replace the default text with the following query.

SQL

```
SELECT
    'reset' AS command
INTO
    alert
FROM
    temperature TIMESTAMP BY timeCreated
GROUP BY TumblingWindow(second,30)
HAVING Avg(machine.temperature) > 70
```

In this query, the SQL code sends a reset command to the alert output if the average machine temperature in a 30-second window reaches 70 degrees. The reset command has been preprogrammed into the sensor as an action that can be taken.

3. Select Save query.

Configure IoT Edge settings

To prepare your Stream Analytics job to be deployed on an IoT Edge device, you need to associate your Azure Stream Analytics job with a storage account. When you deploy your job, the job definition is exported to the storage account in the form of a container.

1. In your Stream Analytics service under the **Settings** menu, select **Storage account settings**.
2. Choose the **Select Blob storage/ADLS Gen 2 from your subscriptions** option.
3. Your Azure storage account automatically shows on the page. If you don't see one, make sure you [create a storage](#). Or if you need to choose a different storage than the one listed in the **Storage account** field, select it from the dropdown menu.
4. Select **Save**, if you had to make any changes.

Deploy the job

You're now ready to deploy the Azure Stream Analytics job on your IoT Edge device.

In this section, you use the **Set Modules** wizard in the Azure portal to create a *deployment manifest*. A deployment manifest is a JSON file that describes all the modules that get deployed to a device. The manifest also shows the container registries that store the module images, how the modules should be managed, and how the modules can communicate with each other. Your IoT Edge device retrieves its

deployment manifest from IoT Hub, then uses the information in it to deploy and configure all of its assigned modules.

For this tutorial, you deploy two modules. The first is **SimulatedTemperatureSensor**, which is a module that simulates a temperature and humidity sensor. The second is your Stream Analytics job. The sensor module provides the stream of data that your job query analyzes.

1. In the Azure portal, navigate to your IoT hub.
2. Select **Devices** under the **Device management** menu, and then select your IoT Edge device to open it.
3. Select **Set modules**.
4. If you previously deployed the SimulatedTemperatureSensor module on this device, it might autopopulate. If it doesn't, add the module with the following steps:
 - a. Select **+ Add** and choose **IoT Edge Module**.
 - b. For the name, type **SimulatedTemperatureSensor**.
 - c. For the image URI, enter **mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.5**.
 - d. Leave the other default settings, then select **Add**.
5. Add your Azure Stream Analytics Edge job with the following steps:
 - a. Select **+ Add** and choose **Azure Stream Analytics Module**.
 - b. Select your subscription and the Azure Stream Analytics Edge job that you created.
 - c. Select **Save**.

Once you save your changes, the details of your Stream Analytics job are published to the storage container that you created.

6. After your Stream Analytics addition finishes deployment, confirm that two new modules appear on your **Set modules** page.

The screenshot shows the 'IoT Edge Modules' section of the Azure portal. At the top, there's a descriptive text about IoT Edge modules. Below it is a header with 'Add' and 'Runtime Settings' buttons. A red box highlights a table listing two modules:

NAME	DESIRED STATUS
SimulatedTemperatureSensor	running
mystreamanalytics	running

On the right side of the table, there are icons for adding a new module and deleting existing ones.

7. Select **Review + create**. The deployment manifest appears.
8. Select **Create**.
9. On your **Set modules** page of your device, after a few minutes, you should see the modules listed and running. Refresh the page if you don't see modules, or wait a few more minutes then refresh it again.

Understand the two new modules

1. From the **Set modules** tab of your device, select your Stream Analytics module name to take you to the **Update IoT Edge Module** page. Here you can update the settings.

The **Settings** tab has the **Image URI** that points to a standard Azure Stream Analytics image. This single image is used for every Stream Analytics module that gets deployed to an IoT Edge device.

The **Module Twin Settings** tab shows the JSON that defines the Azure Stream Analytics (ASA) property called **ASAJobInfo**. The value of that property points to the job definition in your storage container. This property is how the Stream Analytics image is configured with your specific job details.

By default, the Stream Analytics module takes the same name as the job it's based on. You can change the module name on this page if you like, but it's not necessary.

2. Select **Apply** if you made changes or **Cancel** if you didn't make any changes.

Assign routes to your modules

1. On the **Set modules on device:<your-device-name>** page, select **Next: Routes**.
2. On the **Routes** tab, you define how messages are passed between modules and the IoT Hub. Messages are constructed using name and value pairs.

Add the route names and values with the pairs shown in following table. Replace instances of `{moduleName}` with the name of your Azure Stream Analytics module. This module should be the same name you see in the modules list of your device on the **Set modules** page, as shown in the Azure portal.

Modules	IoT Edge hub connections	Deployments and Configurations	
Name	Type	Specified in Deployment	Reported by Device
\$edgeAgent	IoT Edge System Module	✓ Yes	✓ Yes
\$edgeHub	IoT Edge System Module	✓ Yes	✓ Yes
SimulatedTemperatureSensor	IoT Edge Custom Module	✓ Yes	✓ Yes
mystreamanalytics	IoT Edge Custom Module	✓ Yes	✓ Yes

[Expand table](#)

Name	Value
telemetryToCloud	FROM /messages/modules/SimulatedTemperatureSensor/* INTO \$upstream
alertsToCloud	FROM /messages/modules/{moduleName}/* INTO \$upstream
alertsToReset	FROM /messages/modules/{moduleName}/* INTO BrokeredEndpoint("/modules/SimulatedTemperatureSensor/inputs/control")
telemetryToAsa	FROM /messages/modules/SimulatedTemperatureSensor/* INTO BrokeredEndpoint("/modules/{moduleName}/inputs/temperature")

The routes you declare here define the flow of data through the IoT Edge device. The telemetry data from SimulatedTemperatureSensor are sent to IoT Hub and to the **temperature** input that was configured in the Stream Analytics job. The **alert** output messages are sent to IoT Hub and to the SimulatedTemperatureSensor module to trigger the reset command.

3. Select **Next: Review + Create**.
4. In the **Review + Create** tab, you can see how the information you provided in the wizard is converted into a JSON deployment manifest.
5. When you're done reviewing the manifest, select **Create** to finish setting your module.

View data

Now you can go to your IoT Edge device to see the interaction between the Azure Stream Analytics module and the SimulatedTemperatureSensor module.

Note

If you're using a virtual machine for a device, you can use the [Azure Cloud Shell](#) to directly access all Azure authenticated services.

1. Check that all the modules are running in Docker:

```
cmd/sh
```

```
iotedge list
```

2. View all system logs and metrics data. Replace `{moduleName}` with the name of your Azure Stream Analytics module:

```
cmd/sh
```

```
iotedge logs -f {moduleName}
```

3. See how the reset command affects the SimulatedTemperatureSensor by viewing the sensor logs:

```
cmd/sh
```

```
iotedge logs SimulatedTemperatureSensor
```

You can watch the machine's temperature gradually rise until it reaches 70 degrees for 30 seconds. Then the Stream Analytics module triggers a reset, and the machine temperature drops back to 21.

```
53767901}, "ambient": {"temperature": 20.9705382727415, "humidity": 26}, "timeCreated": "2023-03-13T22:17:19.0062666Z"}]}  
03/13/2023 22:17:24> Sending message: 110, Body: [{"machine": {"temperature": 73.92172958651635, "pressure": 7.029057800995533}, "ambient": {"temperature": 21.298097669518597, "humidity": 24}, "timeCreated": "2023-03-13T22:17:24.0118128Z"}]  
03/13/2023 22:17:29> Sending message: 111, Body: [{"machine": {"temperature": 73.74821473657998, "pressure": 7.009290286445821}, "ambient": {"temperature": 20.802224154724843, "humidity": 24}, "timeCreated": "2023-03-13T22:17:29.0158278Z"}]  
03/13/2023 22:17:34> Sending message: 112, Body: [{"machine": {"temperature": 74.10920513635926, "pressure": 7.050415775078271}, "ambient": {"temperature": 21.014030030084158, "humidity": 24}, "timeCreated": "2023-03-13T22:17:34.0194862Z"}]  
Received message Body: [{"command": "reset"}]  
Resetting temperature sensor..  
03/13/2023 22:17:39> Sending message: 113, Body: [{"machine": {"temperature": 21.806618147556026, "pressure": 1.091893206683598}, "ambient": {"temperature": 21.068024410199385, "humidity": 24}, "timeCreated": "2023-03-13T22:17:39.0258684Z"}]  
03/13/2023 22:17:44> Sending message: 114, Body: [{"machine": {"temperature": 21.790987712000955, "pressure": 1.0901125241520075}, "ambient": {"temperature": 20.553355685773444, "humidity": 24}, "timeCreated": "2023-03-13T22:17:44.0293172Z"}]  
03/13/2023 22:17:49> Sending message: 115, Body: [{"machine": {"temperature": 22.496551275368105, "pressure": 1.1704931832697842}, "ambient": {"temperature": 20.539996688738462, "humidity": 25}, "timeCreated": "2023-03-13T22:17:49.0352299Z"}]
```

Clean up resources

If you plan to continue to the next recommended article, you can keep the resources and configurations that you created and reuse them. You can also keep using the same IoT Edge device as a test device.

Otherwise, you can delete the local configurations and the Azure resources that you used in this article to avoid charges.

Delete Azure resources

Deleting Azure resources and resource groups is irreversible. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the IoT hub inside an existing resource group that has resources that you want to keep, delete only the IoT hub resource itself, not the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources that are contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can click into each resource to delete them individually.

Next steps

In this tutorial, you configured an Azure Streaming Analytics job to analyze data from your IoT Edge device. You then loaded this Azure Stream Analytics module on your IoT Edge device to process and react to temperature increase locally, and sending the aggregated data stream to the cloud. To see how Azure IoT Edge can create more solutions for your business, continue on to the other tutorials.

[Deploy an Azure Machine Learning model as a module](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Tutorial: Perform image classification at the edge with Custom Vision Service

Article • 02/21/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

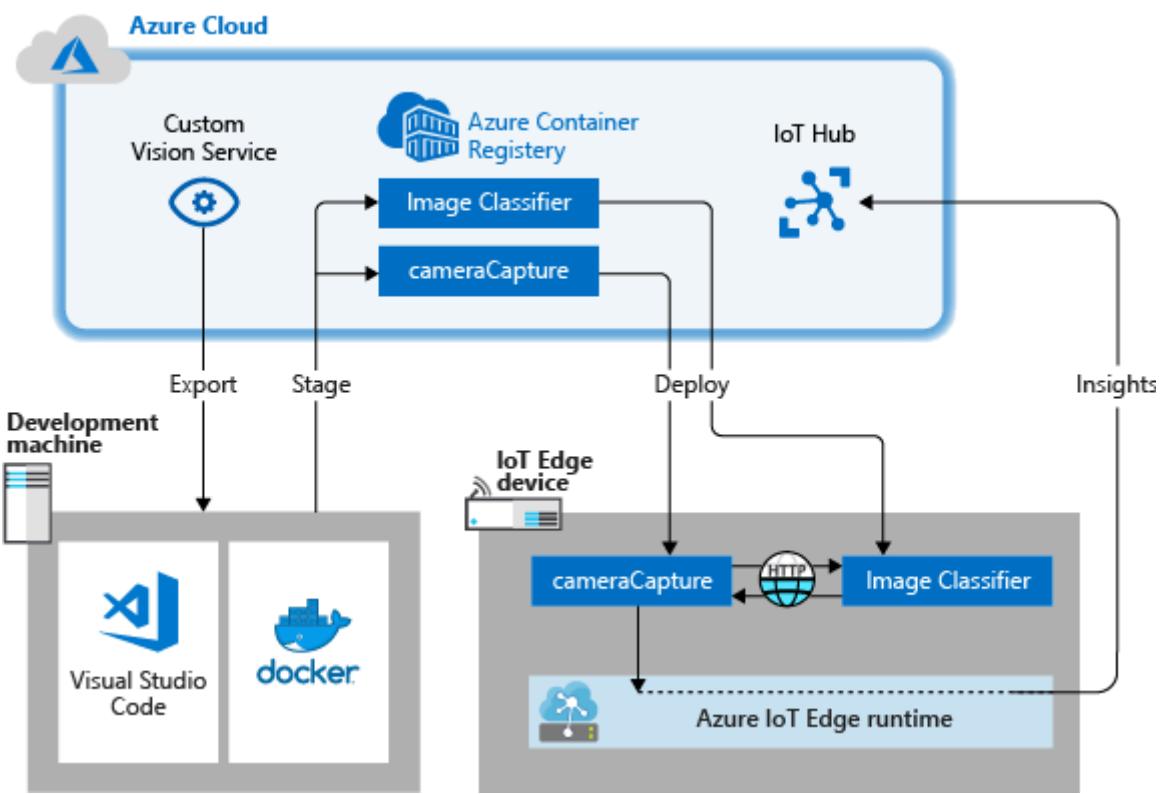
IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge can make your IoT solution more efficient by moving workloads out of the cloud and to the edge. This capability lends itself well to services that process large amounts of data, like computer vision models. [Azure AI Custom Vision](#) lets you build custom image classifiers and deploy them to devices as containers. Together, these two services enable you to find insights from images or video streams without having to transfer all of the data off site first. Custom Vision provides a classifier that compares an image against a trained model to generate insights.

For example, Custom Vision on an IoT Edge device could determine whether a highway is experiencing higher or lower traffic than normal, or whether a parking garage has available parking spots in a row. These insights can be shared with another service to take action.

In this tutorial, you learn how to:

- ✓ Build an image classifier with Custom Vision.
- ✓ Develop an IoT Edge module that queries the Custom Vision web server on your device.
- ✓ Send the results of the image classifier to IoT Hub.



If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

Tip

This tutorial is a simplified version of the [Custom Vision and Azure IoT Edge on a Raspberry Pi 3](#) sample project. This tutorial was designed to run on a cloud VM and uses static images to train and test the image classifier, which is useful for someone just starting to evaluate Custom Vision on IoT Edge. The sample project uses physical hardware and sets up a live camera feed to train and test the image classifier, which is useful for someone who wants to try a more detailed, real-life scenario.

- Configure your environment for Linux container development by completing [Tutorial: Develop IoT Edge modules using Visual Studio Code](#). After completing the tutorial, you should have the following prerequisites in available in your development environment:
 - A free or standard-tier [IoT Hub](#) in Azure.
 - A device running Azure IoT Edge with Linux containers. You can use the quickstarts to set up a [Linux device](#) or [Windows device](#).
 - A container registry, like [Azure Container Registry](#).

- [Visual Studio Code](#) configured with the [Azure IoT Edge](#) and [Azure IoT Hub](#) extensions. The *Azure IoT Edge tools for Visual Studio Code* extension is in [maintenance mode](#).
- Download and install a [Docker compatible container management system](#) on your development machine. Configure it to run Linux containers.
- To develop an IoT Edge module with the Custom Vision service, install the following additional prerequisites on your development machine:
 - [Python](#)
 - [Git](#)
 - [Python extension for Visual Studio Code](#)

Build an image classifier with Custom Vision

To build an image classifier, you need to create a Custom Vision project and provide training images. For more information about the steps that you take in this section, see [How to build a classifier with Custom Vision](#).

Once your image classifier is built and trained, you can export it as a Docker container and deploy it to an IoT Edge device.

Create a new project

1. In your web browser, navigate to the [Custom Vision web page](#).
2. Select **Sign in** and sign in with the same account that you use to access Azure resources.
3. Select **New project**.
4. Create your project with the following values:

[\[\] Expand table](#)

Field	Value
Name	Provide a name for your project, like EdgeTreeClassifier .
Description	Optional project description.
Resource	Select one of your Azure resource groups that includes a Custom Vision Service resource or create new if you haven't yet added one.
Project Types	Classification

Field	Value
Classification Types	Multiclass (single tag per image)
Domains	General (compact)
Export Capabilities	Basic platforms (Tensorflow, CoreML, ONNX, ...)

5. Select **Create project**.

Upload images and train your classifier

Creating an image classifier requires a set of training images and test images.

1. Clone or download sample images from the [Cognitive-CustomVision-Windows](#) ↗ repo onto your local development machine.

```
cmd/sh
```

```
git clone https://github.com/Microsoft/Cognitive-CustomVision-Windows.git
```

2. Return to your Custom Vision project and select **Add images**.
3. Browse to the git repo that you cloned locally, and navigate to the first image folder, **Cognitive-CustomVision-Windows / Samples / Images / Hemlock**. Select all 10 images in the folder and then **Open**.
4. Add the tag **hemlock** to this group of images and press **enter** to apply the tag.
5. Select **Upload 10 files**.

Image upload

X

Add Tags

Uploading

Summary



10 images will be added...

Add some tags to this batch of images...

My Tags

Add a tag and press enter

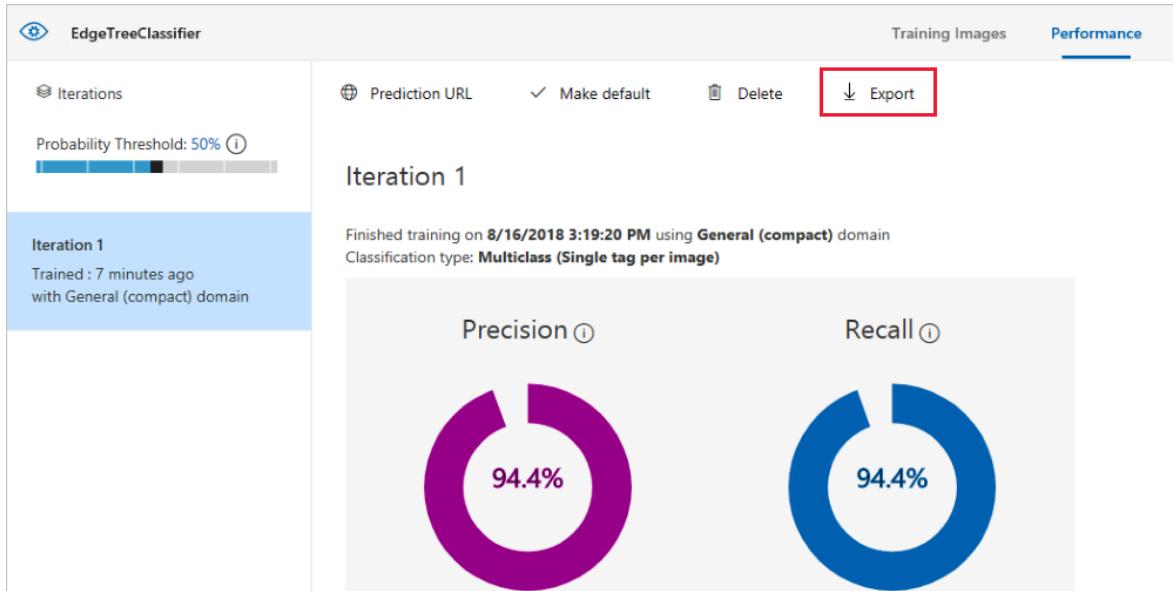
hemlock X

Upload 10 files

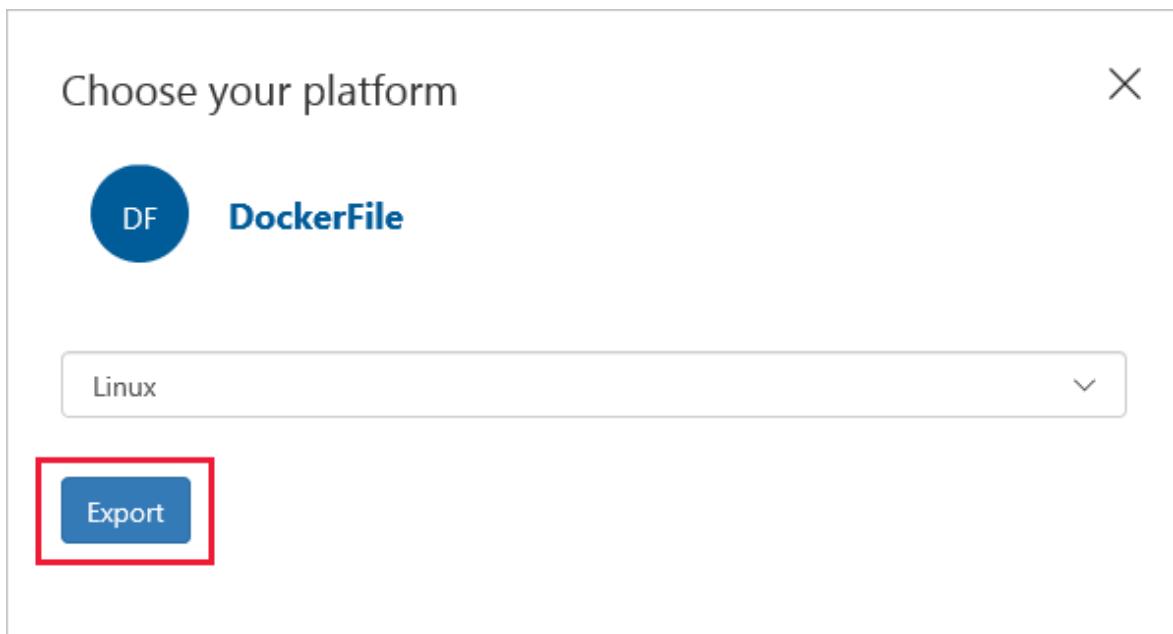
6. When the images are uploaded successfully, select **Done**.
7. Select **Add images** again.
8. Browse to the second image folder, **Cognitive-CustomVision-Windows / Samples / Images / Japanese Cherry**. Select all 10 images in the folder and then **Open**.
9. Add the tag **japanese cherry** to this group of images and press **enter** to apply the tag.
10. Select **Upload 10 files**. When the images are uploaded successfully, select **Done**.
11. When both sets of images are tagged and uploaded, select **Train** to train the classifier.

Export your classifier

1. After training your classifier, select **Export** on the Performance page of the classifier.



2. Select **DockerFile** for the platform.
3. Select **Linux** for the version.
4. Select **Export**.



5. When the export is complete, select **Download** and save the .zip package locally on your computer. Extract all files from the package. You use these files to create an IoT Edge module that contains the image classification server.

When you reach this point, you've finished creating and training your Custom Vision project. You'll use the exported files in the next section, but you're done with the Custom Vision web page.

Create an IoT Edge solution

Now you have the files for a container version of your image classifier on your development machine. In this section, you configure the image classifier container to run as an IoT Edge module. You also create a second module that is deployed alongside the image classifier. The second module posts requests to the classifier and sends the results as messages to IoT Hub.

Create a new solution

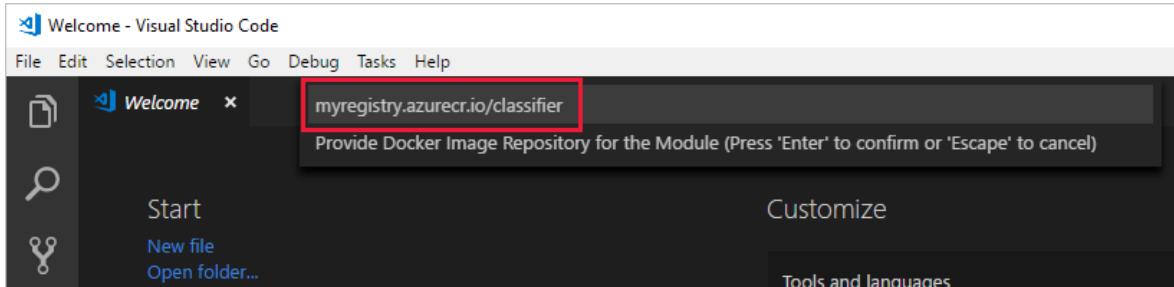
A solution is a logical way of developing and organizing multiple modules for a single IoT Edge deployment. A solution contains code for one or more modules and the deployment manifest that declares how to configure them on an IoT Edge device.

1. In Visual Studio Code, select **View > Command Palette** to open the Visual Studio Code command palette.
2. In the command palette, enter and run the command **Azure IoT Edge: New IoT Edge solution**. In the command palette, provide the following information to create your solution:

[] Expand table

Field	Value
Select folder	Choose the location on your development machine for Visual Studio Code to create the solution files.
Provide a solution name	Enter a descriptive name for your solution, like CustomVisionSolution , or accept the default.
Select module template	Choose Python Module .
Provide a module name	Name your module classifier . It's important that this module name is lowercase. IoT Edge is case-sensitive when referring to modules, and this solution uses a library that formats all requests in lowercase.
Provide Docker image repository for the module	An image repository includes the name of your container registry and the name of your container image. Your container image is prepopulated from the last step. Replace localhost:5000 with the Login server value from your Azure container registry. You can retrieve the Login server from the Overview page of your container registry in the Azure portal.

Field	Value
	The final string looks like <registry name>.azurecr.io/classifier.



The Visual Studio Code window loads your IoT Edge solution workspace.

Add your registry credentials

The environment file stores the credentials for your container registry and shares them with the IoT Edge runtime. The runtime needs these credentials to pull your private images onto the IoT Edge device.

The IoT Edge extension tries to pull your container registry credentials from Azure and populates them in the environment file. Check to see if your credentials are already included. If not, add them now:

1. In the Visual Studio Code explorer, open the .env file.
2. Update the fields with the **username** and **password** values that you copied from your Azure container registry.
3. Save this file.

ⓘ Note

This tutorial uses admin login credentials for Azure Container Registry, which are convenient for development and test scenarios. When you're ready for production scenarios, we recommend a least-privilege authentication option like service principals. For more information, see [Manage access to your container registry](#).

Select your target architecture

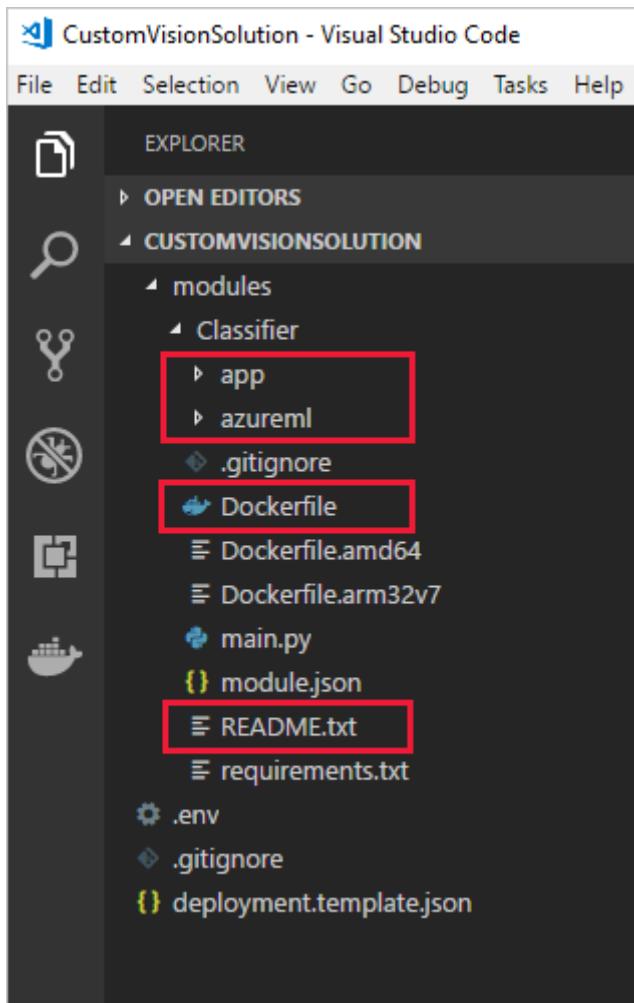
Currently, Visual Studio Code can develop modules for Linux AMD64 and Linux ARM32v7 devices. You need to select which architecture you're targeting with each solution, because the container is built and run differently for each architecture type. The default is Linux AMD64, which is what we use for this tutorial.

1. Open the command palette and search for **Azure IoT Edge: Set Default Target Platform for Edge Solution**, or select the shortcut icon in the side bar at the bottom of the window.
2. In the command palette, select the target architecture from the list of options. For this tutorial, we're using an Ubuntu virtual machine as the IoT Edge device, so keep the default **amd64**.

Add your image classifier

The Python module template in Visual Studio Code contains some sample code that you can run to test IoT Edge. You won't use that code in this scenario. Instead, use the steps in this section to replace the sample code with the image classifier container that you exported previously.

1. In your file explorer, browse to the Custom Vision package that you downloaded and extracted. Copy all the contents from the extracted package. It should be two folders, **app** and **azureml**, and two files, **Dockerfile** and **README**.
2. In your file explorer, browse to the directory where you told Visual Studio Code to create your IoT Edge solution.
3. Open the classifier module folder. If you used the suggested names in the previous section, the folder structure looks like **CustomVisionSolution / modules / classifier**.
4. Paste the files into the **classifier** folder.
5. Return to the Visual Studio Code window. Your solution workspace should now show the image classifier files in the module folder.



6. Open the **module.json** file in the classifier folder.
7. Update the **platforms** parameter to point to the new Dockerfile that you added, and remove all the options besides AMD64, which is the only architecture we're using for this tutorial.

```
JSON

"platforms": {
    "amd64": "./Dockerfile"
}
```

8. Save your changes.

Create a simulated camera module

In a real Custom Vision deployment, you would have a camera providing live images or video streams. For this scenario, you simulate the camera by building a module that sends a test image to the image classifier.

Add and configure a new module

In this section, you add a new module to the same CustomVisionSolution and provide code to create the simulated camera.

1. In the same Visual Studio Code window, use the command palette to run **Azure IoT Edge: Add IoT Edge Module**. In the command palette, provide the following information for your new module:

[] [Expand table](#)

Prompt	Value
Select deployment template file	Select the deployment.template.json file in the CustomVisionSolution folder.
Select module template	Select Python Module
Provide a module name	Name your module cameraCapture
Provide Docker image repository for the module	Replace localhost:5000 with the Login server value for your Azure container registry. The final string looks like <registryname>.azurecr.io/cameracapture.

The Visual Studio Code window loads your new module in the solution workspace, and updates the deployment.template.json file. Now you should see two module folders: classifier and cameraCapture.

2. Open the **main.py** file in the **modules / cameraCapture** folder.
3. Replace the entire file with the following code. This sample code sends POST requests to the image-processing service running in the classifier module. We provide this module container with a sample image to use in the requests. It then packages the response as an IoT Hub message and sends it to an output queue.

Python

```
# Copyright (c) Microsoft. All rights reserved.
# Licensed under the MIT license. See LICENSE file in the project root
# for
# full license information.

import time
import sys
import os
import requests
import json
from azure.iot.device import IoTHubModuleClient, Message
```

```

# global counters
SENT_IMAGES = 0

# global client
CLIENT = None

# Send a message to IoT Hub
# Route output1 to $upstream in deployment.template.json
def send_to_hub(strMessage):
    message = Message(bytarray(strMessage, 'utf8'))
    CLIENT.send_message_to_output(message, "output1")
    global SENT_IMAGES
    SENT_IMAGES += 1
    print( "Total images sent: {}".format(SENT_IMAGES) )

# Send an image to the image classifying server
# Return the JSON response from the server with the prediction result
def sendFrameForProcessing(imagePath, imageProcessingEndpoint):
    headers = {'Content-Type': 'application/octet-stream'}

    with open(imagePath, mode="rb") as test_image:
        try:
            response = requests.post(imageProcessingEndpoint, headers = headers, data = test_image)
            print("Response from classification service: (" +
            str(response.status_code) + ") " + json.dumps(response.json()) + "\n")
        except Exception as e:
            print(e)
            print("No response from classification service")
            return None

    return json.dumps(response.json())

def main(imagePath, imageProcessingEndpoint):
    try:
        print ( "Simulated camera module for Azure IoT Edge. Press Ctrl-C to exit." )

        try:
            global CLIENT
            CLIENT = IoTHubModuleClient.create_from_edge_environment()
        except Exception as iothub_error:
            print ( "Unexpected error {} from IoTHub".format(iothub_error) )
            return

        print ( "The sample is now sending images for processing and will indefinitely." )

        while True:
            classification = sendFrameForProcessing(imagePath,
imageProcessingEndpoint)
            if classification:
                send_to_hub(classification)
                time.sleep(10)

```

```

        except KeyboardInterrupt:
            print ( "IoT Edge module sample stopped" )

if __name__ == '__main__':
    try:
        # Retrieve the image location and image classifying server
        endpoint from container environment
        IMAGE_PATH = os.getenv('IMAGE_PATH', "")
        IMAGE_PROCESSING_ENDPOINT =
os.getenv('IMAGE_PROCESSING_ENDPOINT', "")
    except ValueError as error:
        print ( error )
        sys.exit(1)

    if ((IMAGE_PATH and IMAGE_PROCESSING_ENDPOINT) != ""):
        main(IMAGE_PATH, IMAGE_PROCESSING_ENDPOINT)
    else:
        print ( "Error: Image path or image-processing endpoint
missing" )

```

4. Save the **main.py** file.

5. Open the **requirements.txt** file.

6. Add a new line for a library to include in the container.

Text

requests

7. Save the **requirements.txt** file.

Add a test image to the container

Instead of using a real camera to provide an image feed for this scenario, we're going to use a single test image. A test image is included in the GitHub repo that you downloaded for the training images earlier in this tutorial.

1. Navigate to the test image, located at **Cognitive-CustomVision-Windows / Samples / Images / Test.**

2. Copy **test_image.jpg**

3. Browse to your IoT Edge solution directory and paste the test image in the **modules / cameraCapture** folder. The image should be in the same folder as the **main.py** file that you edited in the previous section.

4. In Visual Studio Code, open the **Dockerfile.amd64** file for the cameraCapture module.

5. After the line that establishes the working directory, `WORKDIR /app`, add the following line of code:

```
Dockerfile  
ADD ./test_image.jpg .
```

6. Save the Dockerfile.

Prepare a deployment manifest

So far in this tutorial you've trained a Custom Vision model to classify images of trees, and packaged that model up as an IoT Edge module. Then, you created a second module that can query the image classification server and report its results back to IoT Hub. Now, you're ready to create the deployment manifest that will tell an IoT Edge device how to start and run these two modules together.

The IoT Edge extension for Visual Studio Code provides a template in each IoT Edge solution to help you create a deployment manifest.

1. Open the **deployment.template.json** file in the solution folder.
2. Find the **modules** section, which should contain three modules: the two that you created, classifier and cameraCapture, and a third that's included by default, SimulatedTemperatureSensor.
3. Delete the **SimulatedTemperatureSensor** module with all of its parameters. This module is included to provide sample data for test scenarios, but we don't need it in this deployment.
4. If you named the image classification module something other than **classifier**, check the name now and ensure that it's all lowercase. The cameraCapture module calls the classifier module using a requests library that formats all requests in lowercase, and IoT Edge is case-sensitive.
5. Update the **createOptions** parameter for the cameraCapture module with the following JSON. This information creates environment variables in the module container that are retrieved in the main.py process. By including this information in the deployment manifest, you can change the image or endpoint without having to rebuild the module image.

JSON

```
"createOptions": "{\"Env\":\n    [\"IMAGE_PATH=test_image.jpg\", \"IMAGE_PROCESSING_ENDPOINT=http://classifier/image\"]}"
```

If you named your Custom Vision module something other than *classifier*, update the image-processing endpoint value to match.

6. At the bottom of the file, update the **routes** parameter for the \$edgeHub module. You want to route the prediction results from cameraCapture to IoT Hub.

JSON

```
"routes": {\n    "cameraCaptureToIoTHub": "FROM\n/messages/modules/cameraCapture/outputs/* INTO $upstream"\n},
```

If you named your second module something other than *cameraCapture*, update the route value to match.

7. Save the **deployment.template.json** file.

Build and push your IoT Edge solution

With both modules created and the deployment manifest template configured, you're ready to build the container images and push them to your container registry.

Once the images are in your registry, you can deploy the solution to an IoT Edge device. You can set modules on a device through the IoT Hub, but you can also access your IoT Hub and devices through Visual Studio Code. In this section, you set up access to your IoT Hub then use Visual Studio Code to deploy your solution to your IoT Edge device.

First, build and push your solution to your container registry.

1. Open the Visual Studio Code integrated terminal by selecting **View > Terminal**.
2. Sign in to Docker by entering the following command in the terminal. Sign in with the username, password, and login server from your Azure container registry. You can retrieve these values from the **Access keys** section of your registry in the Azure portal.

Bash

```
docker login -u <ACR username> -p <ACR password> <ACR login server>
```

You may receive a security warning recommending the use of `--password-stdin`. While that best practice is recommended for production scenarios, it's outside the scope of this tutorial. For more information, see the [docker login](#) reference.

3. In the Visual Studio Code explorer, right-click the `deployment.template.json` file and select **Build and Push IoT Edge solution**.

The build and push command starts three operations. First, it creates a new folder in the solution called `config` that holds the full deployment manifest, which is built out of information in the deployment template and other solution files. Second, it runs `docker build` to build the container image based on the appropriate dockerfile for your target architecture. Then, it runs `docker push` to push the image repository to your container registry.

This process may take several minutes the first time, but is faster the next time that you run the commands.

Deploy modules to device

Use the Visual Studio Code explorer and the Azure IoT Edge extension to deploy the module project to your IoT Edge device. You already have a deployment manifest prepared for your scenario, the `deployment.amd64.json` file in the config folder. All you need to do now is select a device to receive the deployment.

Make sure that your IoT Edge device is up and running.

1. In the Visual Studio Code explorer, under the **Azure IoT Hub** section, expand **Devices** to see your list of IoT devices.
2. Right-click the name of your IoT Edge device, then select **Create Deployment for Single Device**.
3. Select the `deployment.amd64.json` file in the `config` folder and then **Select Edge Deployment Manifest**. Don't use the `deployment.template.json` file.
4. Under your device, expand **Modules** to see a list of deployed and running modules. Select the refresh button. You should see the new `classifier` and `cameraCapture` modules running along with the `$edgeAgent` and `$edgeHub`.

You can also check to see that all the modules are up and running on your device itself. On your IoT Edge device, run the following command to see the status of the modules.

```
Bash
```

```
iotedge list
```

It may take a few minutes for the modules to start. The IoT Edge runtime needs to receive its new deployment manifest, pull down the module images from the container runtime, then start each new module.

View classification results

There are two ways to view the results of your modules, either on the device itself as the messages are generated and sent, or from Visual Studio Code as the messages arrive at IoT Hub.

From your device, view the logs of the cameraCapture module to see the messages being sent and the confirmation that they were received by IoT Hub.

```
Bash
```

```
iotedge logs cameraCapture
```

For example, you should see output like the following:

```
Output
```

```
admin@vm:~$ iotedge logs cameraCapture
Simulated camera module for Azure IoT Edge. Press Ctrl-C to exit.
The sample is now sending images for processing and will indefinitely.
Response from classification service: (200) {"created": "2023-07-
13T17:38:42.940878", "id": "", "iteration": "", "predictions":
[{"boundingBox": null, "probability": 1.0, "tagId": "", "tagName":
"hemlock"}], "project": ""}

Total images sent: 1
Response from classification service: (200) {"created": "2023-07-
13T17:38:53.444884", "id": "", "iteration": "", "predictions":
[{"boundingBox": null, "probability": 1.0, "tagId": "", "tagName":
"hemlock"}], "project": ""}
```

You can also view messages from Visual Studio Code. Right-click the name of your IoT Edge device and select **Start Monitoring Built-in Event Endpoint**.

```
Output
```

```
[IoTHubMonitor] [2:43:36 PM] Message received from [vision-device/cameraCapture]:  
{  
    "created": "2023-07-13T21:43:35.697782",  
    "id": "",  
    "iteration": "",  
    "predictions": [  
        {  
            "boundingBox": null,  
            "probability": 1,  
            "tagId": "",  
            "tagName": "hemlock"  
        }  
    ],  
    "project": ""  
}
```

ⓘ Note

Initially, you may see connection errors in the output from the cameraCapture module. This is due to the delay between modules being deployed and starting.

The cameraCapture module automatically reattempts connection until successful. After successful connection, you see the expected image classification messages.

The results from the Custom Vision module that are sent as messages from the cameraCapture module, include the probability that the image is of either a hemlock or cherry tree. Since the image is hemlock, you should see the probability as 1.0.

Clean up resources

If you plan to continue to the next recommended article, you can keep the resources and configurations that you created and reuse them. You can also keep using the same IoT Edge device as a test device.

Otherwise, you can delete the local configurations and the Azure resources that you used in this article to avoid charges.

Delete Azure resources

Deleting Azure resources and resource groups is irreversible. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the IoT hub

inside an existing resource group that has resources that you want to keep, delete only the IoT hub resource itself, not the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources that are contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can click into each resource to delete them individually.

Next steps

In this tutorial, you trained a Custom Vision model and deployed it as a module onto an IoT Edge device. Then you built a module that can query the image classification service and report its results back to IoT Hub.

Continue to the next tutorials to learn about other ways that Azure IoT Edge can help you turn data into business insights at the edge.

[Store data at the edge with SQL Server databases](#)

Tutorial: Store data at the edge with SQL Server databases

Article • 07/08/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Deploy a SQL Server module to store data on a device running Azure IoT Edge with Linux containers.

Use Azure IoT Edge and SQL Server to store and query data at the edge. Azure IoT Edge has basic storage capabilities to cache messages if a device goes offline, and then forward them when the connection is reestablished. However, you may want more advanced storage capabilities, like being able to query data locally. Your IoT Edge devices can use local databases to perform more complex computing without having to maintain a connection to IoT Hub.

This article provides instructions for deploying a SQL Server database to an IoT Edge device. Azure Functions, running on the IoT Edge device, structures the incoming data then sends it to the database. The steps in this article can also be applied to other databases that work in containers, like MySQL or PostgreSQL.

In this tutorial, you learn how to:

- ✓ Use Visual Studio Code to create an Azure Function
- ✓ Deploy a SQL database to your IoT Edge device
- ✓ Use Visual Studio Code to build modules and deploy them to your IoT Edge device
- ✓ View generated data

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Prerequisites

Before beginning this tutorial, you should have gone through the previous tutorial to set up your development environment for Linux container development: [Develop Azure IoT Edge modules using Visual Studio Code](#). By completing that tutorial, you should have the following prerequisites in place:

- A free or standard-tier [IoT Hub](#) in Azure.
- An AMD64 device running Azure IoT Edge with Linux containers. You can use the quickstarts to set up a [Linux device](#) or [Windows device](#).
 - ARM devices, like Raspberry Pis, cannot run SQL Server. If you want to use SQL on an ARM device, you can use [Azure SQL Edge](#).
- A container registry, like [Azure Container Registry](#).
- [Visual Studio Code](#) configured with the [Azure IoT Edge](#) and [Azure IoT Hub](#) extensions. The *Azure IoT Edge tools for Visual Studio Code* extension is in [maintenance mode](#).
- Download and install a [Docker compatible container management system](#) on your development machine. Configure it to run Linux containers.

This tutorial uses an Azure Functions module to send data to the SQL Server. To develop an IoT Edge module with Azure Functions, install the following additional prerequisites on your development machine:

- [C# for Visual Studio Code \(powered by OmniSharp\)](#) extension for Visual Studio Code.
- [.NET Core SDK](#).

Create a function project

To send data into a database, you need a module that can structure the data properly and then stores it in a table.

Create a new project

The following steps show you how to create an IoT Edge function using Visual Studio Code and the Azure IoT Edge extension.

1. Open Visual Studio Code.
2. Open the Visual Studio Code command palette by selecting **View > Command palette**.
3. In the command palette, type and run the command **Azure IoT Edge: New IoT Edge solution**. In the command palette, provide the following information to

create your solution:

[] Expand table

Field	Value
Select folder	Choose the location on your development machine for Visual Studio Code to create the solution files.
Provide a solution name	Enter a descriptive name for your solution, like SqlSolution , or accept the default.
Select module template	Choose Azure Functions - C# .
Provide a module name	Name your module sqlFunction .
Provide Docker image repository for the module	An image repository includes the name of your container registry and the name of your container image. Your container image is prepopulated from the last step. Replace localhost:5000 with the Login server value from your Azure container registry. You can retrieve the Login server from the Overview page of your container registry in the Azure portal. The final string looks like <registry name>.azurecr.io/sqlfunction.

The Visual Studio Code window loads your IoT Edge solution workspace.

Add your registry credentials

The environment file stores the credentials for your container registry and shares them with the IoT Edge runtime. The runtime needs these credentials to pull your private images onto the IoT Edge device.

The IoT Edge extension tries to pull your container registry credentials from Azure and populates them in the environment file. Check to see if your credentials are already included. If not, add them now:

1. In the Visual Studio Code explorer, open the .env file.
2. Update the fields with the **username** and **password** values that you copied from your Azure container registry.
3. Save this file.

! Note

This tutorial uses admin login credentials for Azure Container Registry, which are convenient for development and test scenarios. When you're ready for production scenarios, we recommend a least-privilege authentication option like service principals. For more information, see [Manage access to your container registry](#).

Select your target architecture

You need to select which architecture you're targeting with each solution, because the container is built and run differently for each architecture type. The default is Linux AMD64.

1. Open the command palette and search for **Azure IoT Edge: Set Default Target Platform for Edge Solution**, or select the shortcut icon in the side bar at the bottom of the window.
2. In the command palette, select the target architecture from the list of options. For this tutorial, we're using an Ubuntu virtual machine as the IoT Edge device, so will keep the default **amd64**.

Update the module with custom code

1. In the Visual Studio Code explorer, open **modules > sqlFunction > sqlFunction.csproj**.
2. Find the group of package references, and add a new one to include **SqlClient**.

```
csproj
<PackageReference Include="System.Data.SqlClient" Version="4.5.1"/>
```

3. Save the **sqlFunction.csproj** file.
4. Open the **sqlFunction.cs** file.
5. Replace the entire contents of the file with the following code:

```
C#
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Azure.Devices.Client;
```

```

using Microsoft.Azure.WebJobs;
using Microsoft.Azure.WebJobs.Extensions.EdgeHub;
using Microsoft.Azure.WebJobs.Host;
using Microsoft.Extensions.Logging;
using Newtonsoft.Json;
using Sql = System.Data.SqlClient;

namespace Functions.Samples
{
    public static class sqlFunction
    {
        [FunctionName("sqlFunction")]
        public static async Task FilterMessageAndSendMessage(
            [EdgeHubTrigger("input1")] Message messageReceived,
            [EdgeHub(OutputName = "output1")] IAsyncCollector<Message>
output,
            ILogger logger)
        {
            const int temperatureThreshold = 20;
            byte[] messageBytes = messageReceived.GetBytes();
            var messageString =
System.Text.Encoding.UTF8.GetString(messageBytes);

            if (!string.IsNullOrEmpty(messageString))
            {
                logger.LogInformation("Info: Received one non-empty
message");
                // Get the body of the message and deserialize it.
                var messageBody =
JsonConvert.DeserializeObject<MessageBody>(messageString);

                //Store the data in SQL db
                const string str = "<sql connection string>";
                using (Sql.SqlConnection conn = new
SqlConnection(str))
                {
                    conn.Open();
                    var insertMachineTemperature = "INSERT INTO
MeasurementsDB.dbo.TemperatureMeasurements VALUES (CONVERT(DATETIME2,'"
+ messageBody.timeCreated + "', 127), 'machine', " +
messageBody.machine.temperature + ");";
                    var insertAmbientTemperature = "INSERT INTO
MeasurementsDB.dbo.TemperatureMeasurements VALUES (CONVERT(DATETIME2,'"
+ messageBody.timeCreated + "', 127), 'ambient', " +
messageBody.ambient.temperature + ");";
                    using (Sql.SqlCommand cmd = new
SqlCommand(insertMachineTemperature + "\n" +
insertAmbientTemperature, conn))
                    {
                        //Execute the command and log the # rows
affected.
                        var rows = await cmd.ExecuteNonQueryAsync();
                        logger.LogInformation($"{rows} rows were
updated");
                    }
                }
            }
        }
    }
}

```

```

        }

        if (messageBody != null &&
messageBody.machine.temperature > temperatureThreshold)
{
    // Send the message to the output as the
temperature value is greater than the threshold.
    using (var filteredMessage = new
Message(messageBytes))
{
    // Copy the properties of the original message
into the new Message object.
    foreach (KeyValuePair<string, string> prop in
messageReceived.Properties)
        {filteredMessage.Properties.Add(prop.Key,
prop.Value);}

    // Add a new property to the message to
indicate it is an alert.
    filteredMessage.Properties.Add("MessageType",
"Alert");

    // Send the message.
    await output.AddAsync(filteredMessage);
    logger.LogInformation("Info: Received and
transferred a message with temperature above the threshold");
}
}
}
}

//Define the expected schema for the body of incoming messages.
class MessageBody
{
    public Machine machine {get; set;}
    public Ambient ambient {get; set;}
    public string timeCreated {get; set;}
}

class Machine
{
    public double temperature {get; set;}
    public double pressure {get; set;}
}

class Ambient
{
    public double temperature {get; set;}
    public int humidity {get; set;}
}
}

```

6. In line 35, replace the string <sql connection string> with the following string. The **Data Source** property references the SQL Server container, which doesn't exist yet. You will create it with the name **SQL** in the next section.

```
Data Source=tcp:sql,1433;Initial Catalog=MeasurementsDB;User  
Id=SA;Password=Strong!Passw0rd;TrustServerCertificate=False;Connection  
Timeout=30;
```

7. Save the `sqlFunction.cs` file.

Add the SQL Server container

A [Deployment manifest](#) declares which modules the IoT Edge runtime will install on your IoT Edge device. You provided the code to make a customized Function module in the previous section, but the SQL Server module is already built and available in the Microsoft Artifact Registry. You just need to tell the IoT Edge runtime to include it, then configure it on your device.

1. In Visual Studio Code, open the command palette by selecting **View > Command palette**.
2. In the command palette, type and run the command **Azure IoT Edge: Add IoT Edge module**. In the command palette, provide the following information to add a new module:

[] [Expand table](#)

Field	Value
Select deployment template file	The command palette highlights the <code>deployment.template.json</code> file in your current solution folder. Select that file.
Select module template	Select Existing Module (Enter Full Image URL) .
Provide a Module Name	Enter <code>sql</code> . This name matches the container name declared in the connection string in the <code>sqlFunction.cs</code> file.
Provide Docker Image for the Module	Enter the following URI to pull the SQL Server container image from the Microsoft Artifact Registry. For Ubuntu based images, use <code>mcr.microsoft.com/mssql/server:latest</code> . For Red Hat Enterprise Linux (RHEL) based images, use <code>mcr.microsoft.com/mssql/rhel/server:latest</code> .

The Azure SQL Edge container image is a lightweight, containerized version of SQL Server that can run on IoT Edge devices. It's optimized for edge scenarios and can run on ARM and AMD64 devices.

3. In your solution folder, open the `deployment.template.json` file.

4. Find the **modules** section. You should see three modules. The module *SimulatedTemperatureSensor* is included by default in new solutions, and provides test data to use with your other modules. The module *sqlFunction* is the module that you initially created and updated with new code. Finally, the module *sql* was imported from the Microsoft Artifact Registry.

 **Tip**

The SQL Server module comes with a default password set in the environment variables of the deployment manifest. Any time that you create a SQL Server container in a production environment, you should [change the default system administrator password](#).

5. Close the `deployment.template.json` file.

Build your IoT Edge solution

In the previous sections, you created a solution with one module, and then added another to the deployment manifest template. The SQL Server module is hosted publicly by Microsoft, but you need to containerize the code in the Functions module. In this section, you build the solution, create container images for the *sqlFunction* module, and push the image to your container registry.

1. In Visual Studio Code, open the integrated terminal by selecting **View > Terminal**.
2. Sign in to your container registry in Visual Studio Code so that you can push your images to your registry. Use the same Azure Container Registry (ACR) credentials that you added to the `.env` file. Enter the following command in the integrated terminal:

```
csh/sh
```

```
docker login -u <ACR username> -p <ACR password> <ACR login server>
```

You might see a security warning recommending the use of the `--password-stdin` parameter. While its use is outside the scope of this article, we recommend following this best practice. For more information, see the [docker login](#) ↗ command reference.

3. In the Visual Studio Code explorer, right-click the `deployment.template.json` file and select **Build and Push IoT Edge solution**.

The build and push command starts three operations. First, it creates a new folder in the solution called **config** that holds the full deployment manifest, which is built out of information in the deployment template and other solution files. Second, it runs `docker build` to build the container image based on the appropriate dockerfile for your target architecture. Then, it runs `docker push` to push the image repository to your container registry.

This process may take several minutes the first time, but is faster the next time that you run the commands.

You can verify that the **sqlFunction** module was successfully pushed to your container registry. In the Azure portal, navigate to your container registry. Select **repositories** and search for **sqlFunction**. The other two modules, **SimulatedTemperatureSensor** and **sql**, won't be pushed to your container registry because their repositories are already in the Microsoft registries.

Deploy the solution to a device

You can set modules on a device through the IoT Hub, but you can also access your IoT Hub and devices through Visual Studio Code. In this section, you set up access to your IoT Hub then use Visual Studio Code to deploy your solution to your IoT Edge device.

1. In the Visual Studio Code explorer, under the **Azure IoT Hub** section, expand **Devices** to see your list of IoT devices.
2. Right-click on the device that you want to target with your deployment and select **Create Deployment for Single Device**.
3. Select the **deployment.amd64.json** file in the **config** folder and then click **Select Edge Deployment Manifest**. Do not use the **deployment.template.json** file.
4. Under your device, expand **Modules** to see a list of deployed and running modules. Click the refresh button. You should see the new **sql** and **sqlFunction** modules running along with the **SimulatedTemperatureSensor** module and the **\$edgeAgent** and **\$edgeHub**.

You can also check to see that all the modules are up and running on your device. On your IoT Edge device, run the following command to see the status of the modules.

```
cmd/sh
```

```
iotedge list
```

It may take a few minutes for the modules to start. The IoT Edge runtime needs to receive its new deployment manifest, pull down the module images from the container runtime, then start each new module.

Create the SQL database

When you apply the deployment manifest to your device, you get three modules running. The SimulatedTemperatureSensor module generates simulated environment data. The sqlFunction module takes the data and formats it for a database. This section guides you through setting up the SQL database to store the temperature data.

Run the following commands on your IoT Edge device. These commands connect to the `sql` module running on your device and create a database and table to hold the temperature data being sent to it.

1. In a command-line tool on your IoT Edge device, connect to your database.

```
Bash
```

```
sudo docker exec -it sql bash
```

2. Open the SQL command tool.

```
Bash
```

```
/opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P 'Strong!Passw0rd'
```

3. Create your database:

```
SQL
```

```
CREATE DATABASE MeasurementsDB
ON
(NAME = MeasurementsDB, FILENAME = '/var/opt/mssql/measurementsdb.mdf')
GO
```

4. Define your table.

```
SQL
```

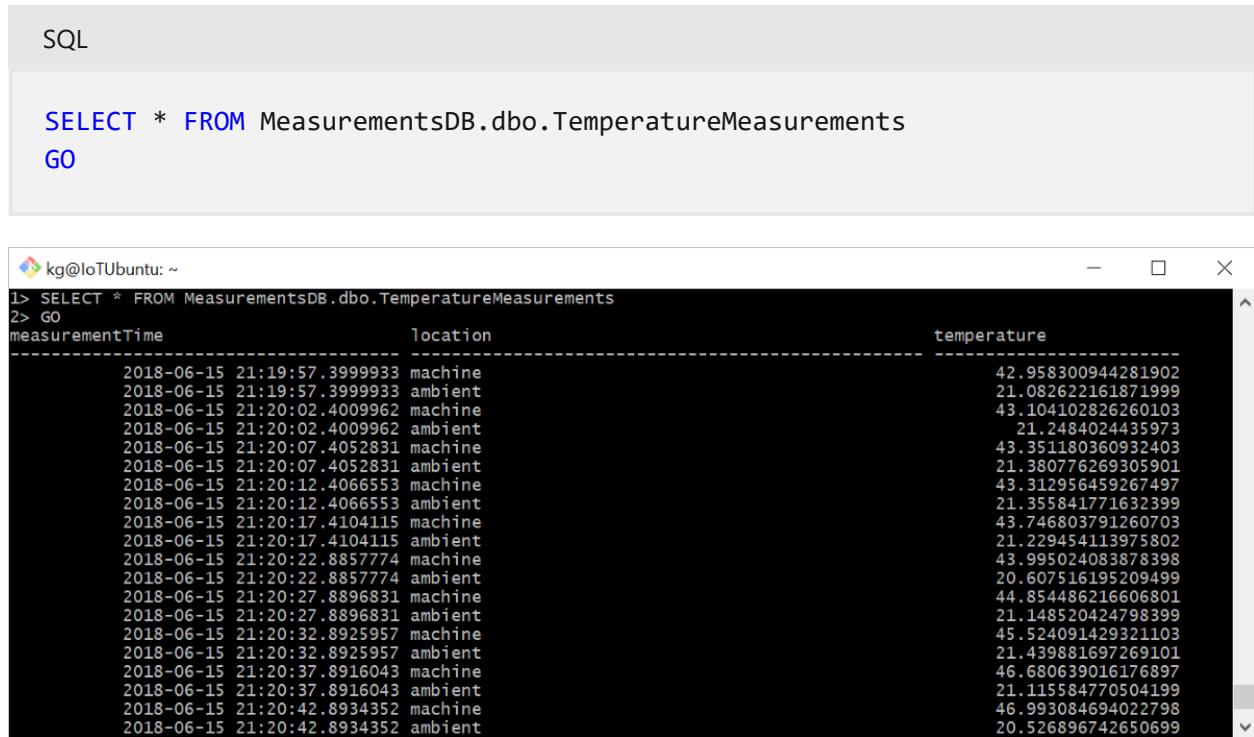
```
CREATE TABLE MeasurementsDB.dbo.TemperatureMeasurements
(measurementTime DATETIME2, location NVARCHAR(50), temperature FLOAT)
GO
```

You can customize your SQL Server docker file to automatically set up your SQL Server to be deployed on multiple IoT Edge devices. For more information, see the [Microsoft SQL Server container demo project](#).

View the local data

Once your table is created, the sqlFunction module starts storing data in a local SQL Server 2017 database on your IoT Edge device.

From inside the SQL command tool, run the following command to view your formatted table data:



The screenshot shows a terminal window titled "SQL". Inside, a command is run to select all data from the "TemperatureMeasurements" table:

```
SELECT * FROM MeasurementsDB.dbo.TemperatureMeasurements
GO
```

The output displays the following data:

measurementTime	location	temperature
2018-06-15 21:19:57.3999933	machine	42.958300944281902
2018-06-15 21:19:57.3999933	ambient	21.082622161871999
2018-06-15 21:20:02.4009962	machine	43.104102826260103
2018-06-15 21:20:02.4009962	ambient	21.2484024435973
2018-06-15 21:20:07.4052831	machine	43.351180360932403
2018-06-15 21:20:07.4052831	ambient	21.380776269305901
2018-06-15 21:20:12.4066553	machine	43.312956459267497
2018-06-15 21:20:12.4066553	ambient	21.355841771632399
2018-06-15 21:20:17.4104115	machine	43.746803791260703
2018-06-15 21:20:17.4104115	ambient	21.229454113975802
2018-06-15 21:20:22.8857774	machine	43.995024083878398
2018-06-15 21:20:22.8857774	ambient	20.607516195209499
2018-06-15 21:20:27.8896831	machine	44.854486216606801
2018-06-15 21:20:27.8896831	ambient	21.148520424798399
2018-06-15 21:20:32.8925957	machine	45.524091429321103
2018-06-15 21:20:32.8925957	ambient	21.439881697269101
2018-06-15 21:20:37.8916043	machine	46.680639016176897
2018-06-15 21:20:37.8916043	ambient	21.115584770504199
2018-06-15 21:20:42.8934352	machine	46.993084694022798
2018-06-15 21:20:42.8934352	ambient	20.526896742650699

Clean up resources

If you plan to continue to the next recommended article, you can keep the resources and configurations that you created and reuse them. You can also keep using the same IoT Edge device as a test device.

Otherwise, you can delete the local configurations and the Azure resources that you created in this article to avoid charges.

Delete Azure resources

Deleting Azure resources and resource groups is irreversible. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the IoT hub

inside an existing resource group that has resources that you want to keep, delete only the IoT hub resource itself, not the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources that are contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can click into each resource to delete them individually.

In this tutorial, you created an Azure Functions module that contains code to filter raw data generated by your IoT Edge device. When you're ready to build your own modules, you can learn more about how to [Develop Azure IoT Edge modules using Visual Studio Code](#).

Next steps

If you want to try another storage method at the edge, read about how to use Azure Blob Storage on IoT Edge.

[Store data at the edge with Azure Blob Storage on IoT Edge](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Tutorial: Create a hierarchy of IoT Edge devices

Article • 06/10/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

You can deploy Azure IoT Edge nodes across networks organized in hierarchical layers. Each layer in a hierarchy is a gateway device that handles messages and requests from devices in the layer beneath it. This configuration is also known as *nested edge*.

You can structure a hierarchy of devices so that only the top layer has connectivity to the cloud, and the lower layers can only communicate with adjacent upstream and downstream layers. This network layering is the foundation of most industrial networks that follow the [ISA-95 standard](#).

This tutorial walks you through creating a hierarchy of IoT Edge devices, deploying IoT Edge runtime containers to your devices, and configuring your devices locally. You do the following tasks:

- ✓ Create and define the relationships in a hierarchy of IoT Edge devices.
- ✓ Configure the IoT Edge runtime on the devices in your hierarchy.
- ✓ Install consistent certificates across your device hierarchy.
- ✓ Add workloads to the devices in your hierarchy.
- ✓ Use the [IoT Edge API Proxy module](#) to securely route HTTP traffic over a single port from your lower layer devices.

Tip

This tutorial includes a mixture of manual and automated steps to provide a showcase of nested IoT Edge features.

If you'd like an entirely automated look at setting up a hierarchy of IoT Edge devices, follow the scripted [Azure IoT Edge for Industrial IoT sample](#). This

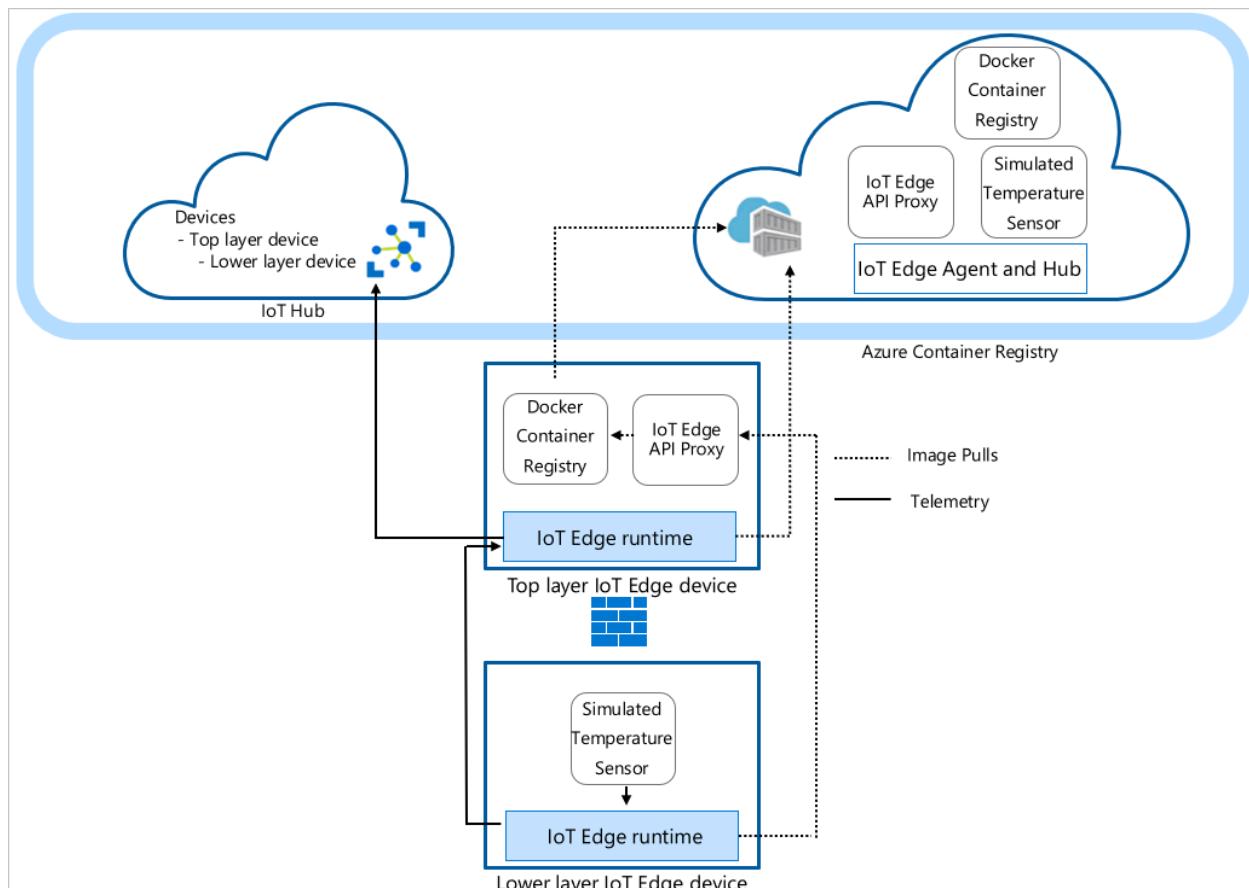
scripted scenario deploys Azure virtual machines as preconfigured devices to simulate a factory environment.

If you'd like an in-depth look at the manual steps to create and manage a hierarchy of IoT Edge devices, see [the how-to guide on IoT Edge device gateway hierarchies](#).

In this tutorial, the following network layers are defined:

- **Top layer:** IoT Edge devices at this layer can connect directly to the cloud.
- **Lower layers:** IoT Edge devices at layers below the top layer can't connect directly to the cloud. They need to go through one or more intermediary IoT Edge devices to send and receive data.

This tutorial uses a two device hierarchy for simplicity. The **top layer device** represents a device at the top layer of the hierarchy that can connect directly to the cloud. This device is referred to as the **parent device**. The **lower layer device** represents a device at the lower layer of the hierarchy that can't connect directly to the cloud. You can add more devices to represent your production environment, as needed. Devices at lower layers are referred to as **child devices**.



① Note

A child device can be a downstream device or a gateway device in a nested topology.

Prerequisites

To create a hierarchy of IoT Edge devices, you need:

- A computer (Windows or Linux) with internet connectivity.
- An Azure account with a valid subscription. If you don't have an [Azure subscription](#), create a [free account](#) before you begin.
- A free or standard tier [IoT Hub](#) in Azure.
- A Bash shell in Azure Cloud Shell using [Azure CLI](#) with the [Azure IoT extension](#) installed. This tutorial uses the [Azure Cloud Shell](#). To see your current versions of the Azure CLI modules and extensions, run `az version`.
- Two Linux devices to configure your hierarchy. If you don't have devices available, you can create Azure virtual machines for each device in your hierarchy using the [IoT Edge Azure Resource Manager template](#). IoT Edge version 1.5 is preinstalled with this Resource Manager template. If you're installing IoT Edge on your own devices, see [Install Azure IoT Edge for Linux](#) or [Update IoT Edge](#).
- To simplify network communication between devices, the virtual machines should be on the same virtual network or use virtual network peering.
- Make sure that the following ports are open inbound for all devices except the lowest layer device: 443, 5671, 8883:
 - 443: Used between parent and child edge hubs for REST API calls and to pull docker container images.
 - 5671, 8883: Used for AMQP and MQTT.

For more information, see [how to open ports to a virtual machine with the Azure portal](#).

Tip

You use the SSH handle and either the FQDN or IP address of each virtual machine for configuration in later steps, so keep track of this information. You can find the IP address and FQDN on the Azure portal. For the IP address, navigate to your list of virtual machines and note the **Public IP address** field. For the FQDN, go to each virtual machine's *overview* page and look for the

DNS name field. For the SSH handle, go to each virtual machine's *connect* page.

Create your IoT Edge device hierarchy

IoT Edge devices make up the layers of your hierarchy. This tutorial creates a hierarchy of two IoT Edge devices: the *top layer device* and the *lower layer device*. You can create more downstream devices as needed.

To create and configure your hierarchy of IoT Edge devices, you use the [az iot edge devices create](#) Azure CLI command. The command simplifies the configuration of the hierarchy by automating and condensing several steps:

- Creates devices in your IoT Hub
- Sets the parent-child relationships to authorize communication between devices
- Applies the deployment manifest to each device
- Generates a chain of certificates for each device to establish secure communication between them
- Generates configuration files for each device

Create device configuration

You create a group of nested edge devices with containing a parent device with one child device. In this tutorial, we use basic sample deployment manifests. For other scenario examples, review the [configuration example templates](#).

1. Before you use the [az iot edge devices create](#) command, you need to define the deployment manifest for the top layer and lower layer devices. Download the [deploymentTopLayer.json](#) sample file to your local machine.

The top layer device deployment manifest defines the [IoT Edge API Proxy](#) module and declares the [route](#) from the lower layer device to IoT Hub.

2. Download the [deploymentLowerLayer.json](#) sample file to your local machine.

The lower layer device deployment manifest includes the simulated temperature sensor module and declares the [route](#) to the top layer device. You can see within **systemModules** section that the runtime modules are set to pull from **\$upstream:443**, instead of **mcr.microsoft.com**. The *lower layer device* sends Docker image requests the *IoT Edge API Proxy* module on port 443, as it can't directly pull the images from the cloud. The other module deployed to the *lower layer device*,

the *Simulated Temperature Sensor* module, also makes its image request to `$upstream:443`.

For more information on how to create a lower layer deployment manifest, see [Connect Azure IoT Edge devices to create a hierarchy](#).

3. In the [Azure Cloud Shell](#), use the `az iot edge devices create` Azure CLI command to create devices in IoT Hub and configuration bundles for each device in your hierarchy. Replace the following placeholders with the appropriate values:

[+] [Expand table](#)

Placeholder	Description
<code><hub-name></code>	The name of your IoT Hub.
<code><config-bundle-output-path></code>	The folder path where you want to save the configuration bundles.
<code><parent-device-name></code>	The <i>top layer</i> parent device ID name.
<code><parent-deployment-manifest></code>	The parent device deployment manifest file.
<code><parent-fqdn-or-ip></code>	Parent device fully qualified domain name (FQDN) or IP address.
<code><child-device-name></code>	The <i>lower layer</i> child device ID name.
<code><child-deployment-manifest></code>	The child device deployment manifest file.
<code><child-fqdn-or-ip></code>	Child device fully qualified domain name (FQDN) or IP address.

Azure CLI

```
az iot edge devices create \
--hub-name <hub-name> \
--output-path <config-bundle-output-path> \
--default-edge-agent "mcr.microsoft.com/azureiotedge-agent:1.5" \
--device id=<parent-device-name> \
    deployment=<parent-deployment-manifest> \
    hostname=<parent-fqdn-or-ip> \
--device id=child-1 \
    parent=parent-1 \
    deployment=<child-deployment-manifest> \
    hostname=<child-fqdn-or-ip>
```

For example, the following command creates a hierarchy of two IoT Edge devices in IoT Hub. A top layer device named *parent-1* and a lower layer device named *child-1**. The command saves the configuration bundles for each device in the *output* directory. The command also generates self-signed test certificates and includes them in the configuration bundle. The configuration bundles are installed on each device using an install script.

Azure CLI

```
az iot edge devices create \
--hub-name my-iot-hub \
--output-path ./output \
--default-edge-agent "mcr.microsoft.com/azureiotedge-agent:1.5" \
--device id=parent-1 \
  deployment=./deploymentTopLayer.json \
  hostname=10.0.0.4 \
--device id=child-1 \
  parent=parent-1 \
  deployment=./deploymentLowerLayer.json \
  hostname=10.1.0.4
```

After running the command, you can find the device configuration bundles in the output directory. For example:

Output

```
PS C:\nested-edge\output> dir
```

```
Directory: C:\nested-edge\output
```

Mode	LastWriteTime	Length	Name
----	-----	-----	-----
-a---	4/10/2023 4:12 PM	7192	child-1.tgz
-a---	4/10/2023 4:12 PM	6851	parent-1.tgz

You can use your own certificates and keys passed as arguments to the command or create a more complex device hierarchy. For more information about creating nested devices using the *az* command, see [az iot edge devices create](#). If you're unfamiliar with how certificates are used in a gateway scenario, see the how-to guide's [certificate section](#).

In this tutorial, you use inline arguments to create the devices and configuration bundles. You can also use a configuration file in YAML or JSON format. For a sample configuration file, see the example [sample_devices_config.yaml](#).

Configure the IoT Edge runtime

In addition to the provisioning of your devices, the configuration steps establish trusted communication between the devices in your hierarchy using the certificates you created earlier. The steps also begin to establish the network structure of your hierarchy. The top layer device maintains internet connectivity, allowing it to pull images for its runtime from the cloud, while lower layer devices route through the top layer device to access these images.

To configure the IoT Edge runtime, you need to apply the configuration bundles to your devices. The configurations differ between the *top layer device* and a *lower layer device*, so be mindful of the device configuration file you're applying to each device.

1. Copy each configuration bundle archive file to its corresponding device. You can use a USB drive, a service like [Azure Key Vault](#), or with a function like [Secure file copy](#). Choose one of these methods that best matches your scenario.

For example, to send the *parent-1* configuration bundle to the home directory on the *parent-1* VM, you could use a command like the following example:

```
Bash
```

```
scp ./output/parent-1.tgz admin@parent-1-vm.westus.cloudapp.azure.com:~
```

2. On each device, extract the configuration bundle archive. For example, use the *tar* command to extract the *parent-1* archive file:

```
Bash
```

```
tar -xzf ./parent-1.tgz
```

3. Set execute permission for the install script.

```
Bash
```

```
chmod +x install.sh
```

4. On each device, apply the configuration bundle to the device using root permission:

```
Bash
```

```
sudo ./install.sh
```

```
azureuser@vm-ozmw2lw145kx6:~$ sudo ./install.sh
Enter the hostname to use: 52.254.22.69
Enter the parent hostname to use: 52.252.5.170
Updating certificates in /etc/ssl/certs...
0 added, 0 removed; done.
Running hooks in /etc/ca-certificates/update.d...
done.
Note: Symmetric key will be written to /var/secrets/aziot/keyd/device-id
Azure IoT Edge has been configured successfully!

Restarting service for configuration to take effect...
Stopping aziot-edged.service...Stopped!
Stopping aziot-identityd.service...Stopped!
Stopping aziot-keyd.service...Stopped!
Stopping aziot-certd.service...Stopped!
Stopping aziot-tpmd.service...Stopped!
Starting aziot-edged.mgmt.socket...Started!
Starting aziot-edged.workload.socket...Started!
Starting aziot-identityd.socket...Started!
Starting aziot-keyd.socket...Started!
Starting aziot-certd.socket...Started!
Starting aziot-tpmd.socket...Started!
Starting aziot-edged.service...Started!
Done.
To check the edge runtime status, run 'iotedge system status'. To validate the configuration, run 'sudo iotedge check'
```

If you want a closer look at what modifications are being made to your device's configuration file, see [Connect Azure IoT Edge devices together to create a hierarchy](#).

To verify your devices are configured correctly, run the configuration and connectivity checks on your devices.

Bash

```
sudo iotedge check
```

Output

```
admin@child-1-vm:~$ sudo iotedge check

Configuration checks (aziot-identity-service)
-----
✓ keyd configuration is well-formed - OK
✓ certd configuration is well-formed - OK
✓ tpmd configuration is well-formed - OK
✓ identityd configuration is well-formed - OK
✓ daemon configurations up-to-date with config.toml - OK
✓ identityd config toml file specifies a valid hostname - OK
✓ host time is close to reference time - OK
✓ preloaded certificates are valid - OK
✓ keyd is running - OK
✓ certd is running - OK
✓ identityd is running - OK
✓ read all preloaded certificates from the Certificates Service - OK
✓ read all preloaded key pairs from the Keys Service - OK
✓ check all EST server URLs utilize HTTPS - OK
✓ ensure all preloaded certificates match preloaded private keys with the
same ID - OK
```

```
Connectivity checks (aziot-identity-service)
-----
```

```
✓ host can connect to and perform TLS handshake with iothub AMQP port - OK
✓ host can connect to and perform TLS handshake with iothub HTTPS /
WebSockets port - OK
✓ host can connect to and perform TLS handshake with iothub MQTT port - OK
```

Configuration checks

```
✓ aziot-edged configuration is well-formed - OK
✓ configuration up-to-date with config.toml - OK
✓ container engine is installed and functional - OK
✓ configuration has correct parent_hostname - OK
✓ configuration has correct URIs for daemon mgmt endpoint - OK
✓ container time is close to host time - OK
!! DNS server - Warning
    Container engine is not configured with DNS server setting, which may
    impact connectivity to IoT Hub.
```

Please see <https://aka.ms/iotedge-prod-checklist-dns> for best practices.

You can ignore this warning if you are setting DNS server per module in
the Edge deployment.

```
!! production readiness: logs policy - Warning
```

Container engine is not configured to rotate module logs which may cause
it run out of disk space.

Please see <https://aka.ms/iotedge-prod-checklist-logs> for best
practices.

You can ignore this warning if you are setting log policy per module in
the Edge deployment.

```
!! production readiness: Edge Agent's storage directory is persisted on the
host filesystem - Warning
```

The edgeAgent module is not configured to persist its /tmp/edgeAgent
directory on the host filesystem.

Data might be lost if the module is deleted or updated.

Please see <https://aka.ms/iotedge-storage-host> for best practices.

```
!! production readiness: Edge Hub's storage directory is persisted on the
host filesystem - Warning
```

The edgeHub module is not configured to persist its /tmp/edgeHub
directory on the host filesystem.

Data might be lost if the module is deleted or updated.

Please see <https://aka.ms/iotedge-storage-host> for best practices.

```
✓ Agent image is valid and can be pulled from upstream - OK
✓ proxy settings are consistent in aziot-edged, aziot-identityd, moby daemon
and config.toml - OK
```

Connectivity checks

```
✓ container on the default network can connect to upstream AMQP port - OK
✓ container on the default network can connect to upstream HTTPS /
WebSockets port - OK
✓ container on the IoT Edge module network can connect to upstream AMQP port
- OK
✓ container on the IoT Edge module network can connect to upstream HTTPS /
WebSockets port - OK
30 check(s) succeeded.
4 check(s) raised warnings. Re-run with --verbose for more details.
2 check(s) were skipped due to errors from other checks. Re-run with --
verbose for more details.
```

On your **top layer device**, expect to see an output with several passing evaluations. You may see some warnings about logs policies and, depending on your network, DNS policies.

Device module deployment

The module deployment for your devices were applied when the devices were created in IoT Hub. The *az iot edge devices create* command applied the deployment JSON files for the top and lower layer devices. After those deployments completed, the **lower layer device** uses the **IoT Edge API Proxy** module to pull its necessary images.

In addition the runtime modules **IoT Edge Agent** and **IoT Edge Hub**, the **top layer device** receives the **Docker registry** module and **IoT Edge API Proxy** module.

The **Docker registry** module points to an existing Azure Container Registry. In this case, `REGISTRY_PROXY_REMOTEURL` points to the Microsoft Container Registry. By default, **Docker registry** listens on port 5000.

The *IoT Edge API Proxy* module routes HTTP requests to other modules, allowing lower layer devices to pull container images or push blobs to storage. In this tutorial, it communicates on port 443 and is configured to send Docker container image pull requests route to your **Docker registry** module on port 5000. Also, any blob storage upload requests route to module `AzureBlobStorageonIoTEdge` on port 11002. For more information about the **IoT Edge API Proxy** module and how to configure it, see the module's [how-to guide](#).

If you'd like a look at how to create a deployment like this through the Azure portal or Azure Cloud Shell, see [top layer device section of the how-to guide](#).

You can view the status of your modules using the command:

Azure CLI

```
az iot hub module-twin show --device-id <edge-device-id> --module-id  
'${edgeAgent}' --hub-name <iot-hub-name> --query "properties.reported.  
[systemModules, modules]"
```

This command outputs all the `edgeAgent` reported properties. Here are some helpful ones for monitoring the status of the device: *runtime status*, *runtime start time*, *runtime last exit time*, *runtime restart count*.

You can also see the status of your modules on the [Azure portal](#). Navigate to the **Devices** section of your IoT Hub to see your devices and modules.

View generated data

The **Simulated Temperature Sensor** module that you pushed generates sample environment data. It sends messages that include ambient temperature and humidity, machine temperature and pressure, and a timestamp.

You can also view these messages through the [Azure Cloud Shell](#):

Azure CLI

```
az iot hub monitor-events -n <iot-hub-name> -d <lower-layer-device-name>
```

For example:

Azure CLI

```
az iot hub monitor-events -n my-iot-hub -d child-1
```

Output

```
{
  "event": {
    "origin": "child-1",
    "module": "simulatedTemperatureSensor",
    "interface": "",
    "component": "",
    "payload": "{\"machine\": {\"temperature\":104.29281270901808,\"pressure\":10.48905461241978},\"ambient\": {\"temperature\":21.086561171611102,\"humidity\":24},\"timeCreated\":\"2023-04-17T21:50:30.1082487Z\"}"
  }
}
```

Troubleshooting

Run the `iotedge check` command to verify the configuration and to troubleshoot errors.

You can run `iotedge check` in a nested hierarchy, even if the downstream machines don't have direct internet access.

When you run `iotedge check` from the lower layer, the program tries to pull the image from the parent through port 443.

The `azureiotedge-diagnostics` value is pulled from the container registry that's linked with the registry module. This tutorial has it set by default to <https://mcr.microsoft.com>:

 Expand table

Name	Value
REGISTRY_PROXY_REMOTEURL	https://mcr.microsoft.com

If you're using a private container registry, make sure that all the images (IoTEdgeAPIProxy, edgeAgent, edgeHub, Simulated Temperature Sensor, and diagnostics) are present in the container registry.

If a downstream device has a different processor architecture from the parent device, you need the appropriate architecture image. You can use a [connected registry](#) or you can specify the correct image for the `edgeAgent` and `edgeHub` modules in the downstream device `config.toml` file. For example, if the parent device is running on an ARM32v7 architecture and the downstream device is running on an AMD64 architecture, you need to specify the matching version and architecture image tag in the downstream device `config.toml` file.

```
toml

[agent.config]
image = "$upstream:443/azureiotedge-agent:1.5.0-linux-amd64"

"systemModules": {
    "edgeAgent": {
        "settings": {
            "image": "$upstream:443/azureiotedge-agent:1.5.0-linux-amd64"
        },
    },
    "edgeHub": {
        "settings": {
            "image": "$upstream:443/azureiotedge-hub:1.5.0-linux-amd64",
        }
    }
}
```

Clean up resources

You can delete the local configurations and the Azure resources that you created in this article to avoid charges.

To delete the resources:

1. Sign in to the [Azure portal](#) and select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can select each resource to delete them individually.

Next steps

In this tutorial, you configured two IoT Edge devices as gateways and set one as the parent device of the other. Then, you pulled a container image onto the downstream device through a gateway using the IoT Edge API Proxy module. See [the how-to guide on the proxy module's use](#) if you want to learn more.

To learn more about using gateways to create hierarchical layers of IoT Edge devices, see the following article.

[Connect Azure IoT Edge devices to create a hierarchy](#)

ⓘ **Note:** The author created this article with assistance from AI. [Learn more](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Tutorial: Configure Enrollment over Secure Transport Server for Azure IoT Edge

Article • 06/10/2024

Applies to: ✓ IoT Edge 1.5 ✓ IoT Edge 1.4

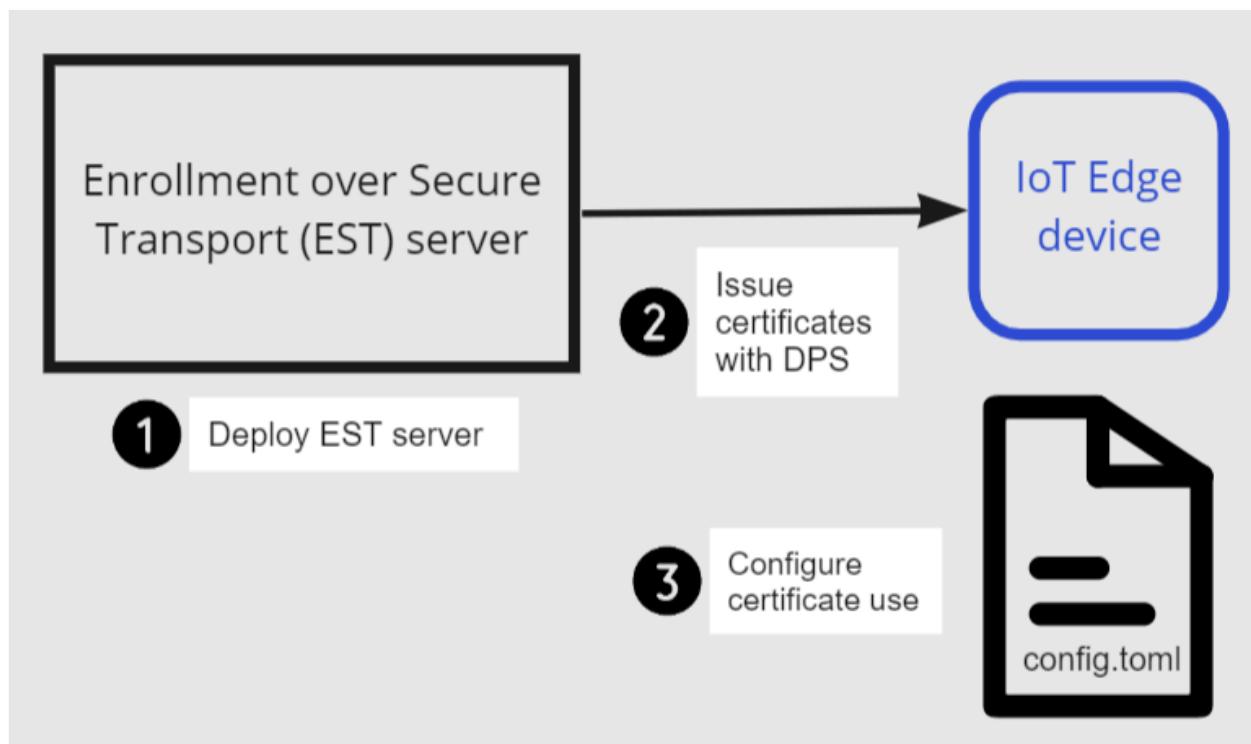
ⓘ Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

With Azure IoT Edge, you can configure your devices to use an Enrollment over Secure Transport (EST) server to manage x509 certificates.

This tutorial walks you through hosting a test EST server and configuring an IoT Edge device for the enrollment and renewal of x509 certificates. In this tutorial, you learn how to:

- ✓ Create and host a test EST server
- ✓ Configure DPS group enrollment
- ✓ Configure device



Prerequisites

- An existing IoT Edge device with the [latest Azure IoT Edge runtime](#) installed. If you need to create a test device, complete [Quickstart: Deploy your first IoT Edge module to a virtual Linux device](#).
- Your IoT Edge device requires Azure IoT Edge runtime 1.2 or later for EST support. Azure IoT Edge runtime 1.3 or later required for EST certificate renewal.
- IoT Hub Device Provisioning Service (DPS) linked to IoT Hub. For information on configuring DPS, see [Quickstart: Set up the IoT Hub Device Provisioning Service with the Azure portal](#).

What is Enrollment over Secure Transport?

Enrollment over Secure Transport (EST) is a cryptographic protocol that automates the issuance of x.509 certificates. It's used for public key infrastructure (PKI) clients, like IoT Edge that need client certificates associated to a Certificate Authority (CA). EST replaces the need for manual certificate management, which can be risky and error-prone.

EST server

For certificate issuance and renewal, you need an EST server accessible to your devices.

 **Important**

For enterprise grade solutions, consider: [GlobalSign IoT Edge Enroll](#) or [DigiCert IoT Device Manager](#).

For testing and development, you can use a test EST server. In this tutorial, we'll create a test EST server.

Run EST server on device

To quickly get started, this tutorial shows the steps to deploy a simple EST server in a container locally on the IoT Edge device. This method is the simplest approach to try it out.

The Dockerfile uses Ubuntu 18.04, a [Cisco library called libest](#), and [sample server code](#). It's configured with the following setting you can change:

- Root CA valid for 20 years

- EST server certificate valid for 10 years
- Set the certificate default-days to 1 to test EST renewal
- EST server runs locally on the IoT Edge device in a container

Caution

Do not use this Dockerfile in production.

1. Connect to the device, for example using SSH, where you've installed IoT Edge.
2. Create a file named `Dockerfile` (case sensitive) and add the sample content using your favorite text editor.

Tip

If you want to host your EST server in Azure Container Instance, change `myestserver.westus.azurecontainer.io` to the DNS name of your EST server. When choosing a DNS name, be aware the DNS label for an Azure Container instance must be at least five characters in length.

Dockerfile

```
# DO NOT USE IN PRODUCTION - Use only for testing #

FROM ubuntu:18.04

RUN apt update && apt install -y apache2-utils git openssl libssl-dev build-essential && \
    git clone https://github.com/cisco/libest.git && cd libest && \
    ./configure --disable-safec && make install && \
    rm -rf /src && apt remove --quiet -y libssl-dev build-essential && \
    apt autoremove -y && apt clean -y && apt autoclean -y && \
    rm -rf /var/lib/apt /tmp/* /var/tmp/*

WORKDIR /libest/example/server/

# Setting the root CA expiration to 20 years
RUN sed -i "s|-days 365|-days 7300 |g" ./createCA.sh

## If you want to host your EST server remotely (for example, an Azure Container Instance),
## change myestserver.westus.azurecontainer.io to the fully qualified DNS name of your EST server
## OR, change the IP address
## and uncomment the corresponding line.
# RUN sed -i "s|DNS.2 = ip6-localhost|DNS.2 =
```

```

myestserver.westus.azurecontainer.io|g" ./ext.cnf
# RUN sed -i "s|IP.2 = ::1|IP.2 = <YOUR EST SERVER IP ADDRESS>|g"
./ext.cnf

# Set EST server certificate to be valid for 10 years
RUN sed -i "s|-keyout \$EST_SERVER_PRIVKEY -subj|-keyout
\$EST_SERVER_PRIVKEY -days 7300 -subj |g" ./createCA.sh

# Create the CA
RUN echo 1 | ./createCA.sh

# Set cert default-days to 1 to show EST renewal
RUN sed -i "s|default_days    = 365|default_days    = 1 |g"
./estExampleCA.cnf

# The EST server listens on port 8085 by default
# Uncomment to change the port to 443 or something else. If changed,
EXPOSE that port instead of 8085.
# RUN sed -i "s|estserver -c|estserver -p 443 -c |g" ./runserver.sh
EXPOSE 8085
CMD ./runserver.sh

```

3. In the directory containing your `Dockerfile`, build your image from the sample `Dockerfile`.

Bash

```
sudo docker build . --tag est
```

4. Start the container and expose the container's port 8085 to port 8085 on the host.

Bash

```
sudo docker run -d -p 8085:8085 est
```

5. Now, your EST server is running and can be reached using `localhost` on port 8085. Verify that it's available by running a command to see its server certificate.

Bash

```
openssl s_client -showcerts -connect localhost:8085
```

6. You should see `-----BEGIN CERTIFICATE-----` midway through the output. Retrieving the certificate verifies that the server is reachable and can present its certificate.

💡 Tip

To run this container in the cloud, build the image and [push the image to Azure Container Registry](#). Then, follow the [quickstart to deploy to Azure Container Instance](#).

Download CA certificate

Each device requires the Certificate Authority (CA) certificate that is associated to a device identity certificate.

1. On the IoT Edge device, create the `/var/aziot/certs` directory if it doesn't exist then change directory to it.

Bash

```
# If the certificate directory doesn't exist, create, set ownership, and
# set permissions
sudo mkdir -p /var/aziot/certs
sudo chown aziotcs:aziotcs /var/aziot/certs
sudo chmod 755 /var/aziot/certs

# Change directory to /var/aziot/certs
cd /var/aziot/certs
```

2. Retrieve the CA certificate from the EST server into the `/var/aziot/certs` directory and name it `cacert.crt.pem`.

Bash

```
openssl s_client -showcerts -verify 5 -connect localhost:8085 <
/dev/null | sudo awk '/BEGIN/,/END/{ if(/BEGIN/){a++};
out="cert" a ".pem"; print >out}' && sudo cp cert2.pem cacert.crt.pem
```

3. Certificates should be owned by the key service user `aziotcs`. Set the ownership to `aziotcs` for all the certificate files and set permissions. For more information about certificate ownership and permissions, see [Permission requirements](#).

Bash

```
# Give aziotcs ownership to certificates
sudo chown -R aziotcs:aziotcs /var/aziot/certs
```

```
# Read and write for aziotcs, read-only for others
sudo find /var/aziot/certs -type f -name "*.*" -exec chmod 644 {} \;
```

Provision IoT Edge device using DPS

Using Device Provisioning Service allows you to automatically issue and renew certificates from an EST server in IoT Edge. When using the tutorial EST server, the identity certificates expire in one day making manual provisioning with IoT Hub impractical since each time the certificate expires the thumbprint must be manually updated in IoT Hub. DPS CA authentication with enrollment group allows the device identity certificates to be renewed without any manual steps.

Upload CA certificate to DPS

1. If you don't have a Device Provisioning Service linked to IoT Hub, see [Quickstart: Set up the IoT Hub Device Provisioning Service with the Azure portal](#).
2. Transfer the `cacert.crt.pem` file from your device to a computer with access to the Azure portal such as your development computer. An easy way to transfer the certificate is to remotely connect to your device, display the certificate using the command `cat /var/aziot/certs/cacert.crt.pem`, copy the entire output, and paste the contents to a new file on your development computer.
3. In the [Azure portal](#), navigate to your instance of IoT Hub Device Provisioning Service.
4. Under **Settings**, select **Certificates**, then **+Add**.

Certificate name * ⓘ
est-ca-cert

Certificate .pem or .cer file. ⓘ
"cacert.crt.pem"

Set certificate status to verified on upload ⓘ

i We'll verify this certificate automatically, with no manual verification steps required. [Learn more](#)

Save

[+] Expand table

Setting	Value
Certificate name	Provide a friendly name for the CA certificate
Certificate .pem or .cer file	Browse to the <code>cacert.crt.pem</code> from the EST server
Set certificate status to verified on upload	Select the checkbox

5. Select **Save**.

Create enrollment group

1. In the [Azure portal](#), navigate to your instance of IoT Hub Device Provisioning Service.
2. Under **Settings**, select **Manage enrollments**.
3. Select **Add enrollment group** then complete the following steps to configure the enrollment.
4. On the **Registration + provisioning** tab, choose the following settings:

Add enrollment group ...

Raj-pnp-dps

Registration + provisioning IoT hubs Device settings Review + create

Attestation

Attestation is the process of verifying a device's identity during registration. Devices must attest their identity using the enrollment's selected attestation mechanism.

Attestation mechanism *

X.509 certificates uploaded to this Device Provisioning Service instance

X.509 certificate settings

Using X.509 certificate attestation, Device Provisioning Service verifies devices' chain of trust against uploaded certificates. Enrollments may specify one or two certificates stored in this Device Provisioning Service instance.

Primary certificate

est-ca-cert

Secondary certificate

Group name

Group name uniquely identifies the enrollment group and is used to find device registration records.

Group name *

EST

Provisioning status

You can enable or disable this enrollment from provisioning and reprovisioning devices.

Enable this enrollment

Reprovision policy

Provisioned devices may trigger requests to be reprovisioned. Reprovision policy specifies whether to reprovision the device and how handle the device's existing state data.

Reprovision policy

Reprovision device and migrate current state

[+] Expand table

Setting	Value
Attestation mechanism	Select X.509 certificates uploaded to this Device Provisioning Service instance
Primary certificate	Choose your certificate from the dropdown list
Group name	Provide a friendly name for this group enrollment
Provisioning status	Select Enable this enrollment checkbox

- On the **IoT hubs** tab, choose your IoT Hub from the list.
- On the **Device settings** tab, select the **Enable IoT Edge on provisioned devices** checkbox.

The other settings aren't relevant to the tutorial. You can accept the default settings.

- Select **Review + create**.

Now that an enrollment exists for the device, the IoT Edge runtime can automatically manage device certificates for the linked IoT Hub.

Configure IoT Edge device

On the IoT Edge device, update the IoT Edge configuration file to use device certificates from the EST server.

1. Open the IoT Edge configuration file using an editor. For example, use the `nano` editor to open the `/etc/aziot/config.toml` file.

```
Bash
```

```
sudo nano /etc/aziot/config.toml
```

2. Add or replace the following sections in the configuration file. These configuration settings use username and password authentication initially to get the device certificate from the EST server. The device certificate is used to authenticate to the EST server for future certificate renewals.

Replace the following placeholder text: `<DPS-ID-SCOPE>` with the **ID Scope** of the DPS linked to the IoT Hub containing the registered device, and `myiotedgedevice` with the device ID registered in Azure IoT Hub. You can find the **ID Scope** value on the DPS [Overview](#) page.

```
Bash
```

```
# DPS provisioning with X.509 certificate
# Replace with ID Scope from your DPS
[provisioning]
source = "dps"
global_endpoint = "https://global.azure-devices-provisioning.net"
id_scope = "<DPS-ID-SCOPE>

[provisioning.attestation]
method = "x509"
registration_id = "myiotedgedevice"

[provisioning.attestation.identity_cert]
method = "est"
common_name = "myiotedgedevice"

# Auto renewal settings for the identity cert
# Available only from IoT Edge 1.3 and above
[provisioning.attestation.identity_cert.auto_renew]
rotate_key = false
threshold = "80%"
retry = "4%"

# Trusted root CA certificate in the global EST options
# Optional if the EST server's TLS certificate is already trusted by
# the system's CA certificates.
[cert_issuance.est]
trusted_certs = [
```

```
        "file:///var/aziot/certs/cacert.crt.pem",
    ]

# The default username and password for libest
# Used for initial authentication to EST server
#
# Not recommended for production
[cert_issuance.est.auth]
username = "estuser"
password = "estpwd"

[cert_issuance.est.urls]
default = "https://localhost:8085/.well-known/est"
```

(!) Note

In this example, IoT Edge uses username and password to authenticate to the EST server *everytime* it needs to obtain a certificate. This method isn't recommended in production because 1) it requires storing a secret in plaintext and 2) IoT Edge should use an identity certificate to authenticate to the EST server too. To modify for production:

- a. Consider using long-lived *bootstrap certificates* that can be stored onto the device during manufacturing [similar to the recommended approach for DPS](#). To see how to configure bootstrap certificate for EST server, see [Authenticate a Device Using Certificates Issued Dynamically via EST](#).
- b. Configure [cert_issuance.est.identity_auto_renew] using the [same syntax](#) as the provisioning certificate auto-renew configuration above.

This way, IoT Edge certificate service uses the bootstrap certificate for initial authentication with EST server, and requests an identity certificate for future EST requests to the same server. If, for some reason, the EST identity certificate expires before renewal, IoT Edge falls back to using the bootstrap certificate.

3. Run `sudo iotedge config apply` to apply the new settings.
4. Run `sudo iotedge check` to verify your IoT Edge device configuration. All **configuration checks** should succeed. For this tutorial, you can ignore production readiness errors and warnings, DNS server warnings, and connectivity checks.
5. Navigate to your device in IoT Hub. Certificate thumbprints have been added to the device automatically using DPS and the EST server.

Home > my-iot-hub >

myiotedgedevice

my-iot-hub

Save Set modules Manage child devices Troubleshoot Device twin Refresh

Device ID	myiotedgedevice
Primary Thumbprint	FF7860926BFEA9A411B3E07A0FC1239F9CC6981706A52BBF952B1357E1481545
Secondary Thumbprint	FF7860926BFEA9A411B3E07A0FC1239F9CC6981706A52BBF952B1357E1481545
IoT Edge Runtime Response	417 -- The device's deployment configuration is not set
Enable connection to IoT Hub	<input checked="" type="radio"/> Enable <input type="radio"/> Disable

ⓘ Note

When you create a new IoT Edge device, it displays the status code 417 -- The device's deployment configuration is not set in the Azure portal. This status is normal, and means that the device is ready to receive a module deployment.

Test certificate renewal

You can immediately reissue the device identity certificates by removing the existing certificates and keys from the device and then applying the IoT Edge configuration. IoT Edge detects the missing files and requests new certificates.

1. On the IoT Edge device, stop the IoT Edge runtime.

```
Bash  
sudo iotedge system stop
```

2. Delete the existing certificates and keys.

```
Bash  
sudo sh -c "rm /var/lib/aziot/certd/certs/*"  
sudo sh -c "rm /var/lib/aziot/keyd/keys/*"
```

3. Apply the IoT Edge configuration to renew certificates.

```
Bash
```

```
sudo iotedge config apply
```

You may need to wait a few minutes for the runtime to start.

4. Navigate to your device in IoT Hub. Certificate thumbprints have been updated.

The screenshot shows the Azure IoT Hub Device Details page for a device named 'myiotedgedevice'. The top navigation bar includes 'Home > my-iot-hub > myiotedgedevice'. Below the device name are several tabs: 'Save' (with a save icon), 'Set modules' (with a gear icon), 'Manage child devices' (with a child device icon), 'Troubleshoot' (with a wrench icon), 'Device twin' (with a gear icon), and 'Refresh' (with a circular arrow icon). The main content area displays the following information:

Device ID	myiotedgedevice
Primary Thumbprint	232A9D6301126ABB4408298E12F537EADE811AF1A8E5BB96EDCC2D2EC05D6BB9
Secondary Thumbprint	232A9D6301126ABB4408298E12F537EADE811AF1A8E5BB96EDCC2D2EC05D6BB9
IoT Edge Runtime Response	417 -- The device's deployment configuration is not set
Enable connection to IoT Hub	<input checked="" type="radio"/> Enable <input type="radio"/> Disable

5. List the certificate files using the command `sudo ls -l`

`/var/lib/aziot/certd/certs`. You should see recent creation dates for the device certificate files.

6. Use the `openssl` command to check the new certificate contents. For example:

Bash

```
sudo openssl x509 -in /var/lib/aziot/certd/certs/deviceid-
bd732105ef89cf8edd2606a5309c8a26b7b5599a4e124a0fe6199b6b2f60e655.cer -
text -noout
```

Replace the device certificate file name (.cer) with your device's certificate file.

You should notice the certificate **Validity** date range has changed.

The following are optional other ways you can test certificate renewal. These checks demonstrate how IoT Edge renews certificates from the EST server when they expire or are missing. After each test, you can verify new thumbprints in the Azure portal and use `openssl` command to verify the new certificate.

1. Try waiting a day for the certificate to expire. The test EST server is configured to create certificates that expire after one day. IoT Edge automatically renews the certificate.

2. Try adjusting the percentage in `threshold` for auto renewal set in `config.toml` (currently set to 80% in the example configuration). For example, set it to `10%` and observe the certificate renewal every ~2 hours.
3. Try adjusting the `threshold` to an integer followed by `m` (minutes). For example, set it to `60m` and observe certificate renewal 1 hours before expiry.

Clean up resources

You can keep the resources and configurations that you created in this tutorial and reuse them. Otherwise, you can delete the local configurations and the Azure resources that you used in this article to avoid charges.

Delete Azure resources

Deleting Azure resources and resource groups is irreversible. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the IoT hub inside an existing resource group that has resources that you want to keep, delete only the IoT hub resource itself, not the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources that are contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can click into each resource to delete them individually.

Next steps

- To use EST server to issue Edge CA certificates, see [example configuration](#).
- Using username and password to bootstrap authentication to EST server isn't recommended for production. Instead, consider using long-lived *bootstrap certificates* that can be stored onto the device during manufacturing [similar to the recommended approach for DPS](#). To see how to configure bootstrap certificate for EST server, see [Authenticate a Device Using Certificates Issued Dynamically via EST](#).
- EST server can be used to issue certificates for all devices in a hierarchy as well. Depending on if you have ISA-95 requirements, it may be necessary to run a chain

of EST servers with one at every layer or use the API proxy module to forward the requests. To learn more, see [Kevin's blog](#).

- For enterprise grade solutions, consider: [GlobalSign IoT Edge Enroll](#) or [DigiCert IoT Device Manager](#)
 - To learn more about certificates, see [Understand how Azure IoT Edge uses certificates](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Tutorial: Configure, connect, and verify an IoT Edge module for a GPU

Article • 05/29/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This tutorial shows you how to build a GPU-enabled virtual machine (VM). From the VM, you'll see how to run an IoT Edge device that allocates work from one of its modules to your GPU.

We'll use the Azure portal, the Azure Cloud Shell, and your VM's command line to:

- Build a GPU-capable VM
- Install the [NVIDIA driver extension](#) on the VM
- Configure a module on an IoT Edge device to allocate work to a GPU

Prerequisites

- Azure account - [create a free account](#)
- Azure IoT Hub - [create an IoT Hub](#)
- Azure IoT Edge device

If you don't already have an IoT Edge device and need to quickly create one, run the following command. Use the [Azure Cloud Shell](#) located in the Azure portal.

Create a new device name for <DEVICE-NAME> and replace the IoT <IOT-HUB-NAME> with your own.

Azure CLI

```
az iot hub device-identity create --device-id <YOUR-DEVICE-NAME> --  
edge-enabled --hub-name <YOUR-IOT-HUB-NAME>
```

For more information on creating an IoT Edge device, see [Quickstart: Deploy your first IoT Edge module to a virtual Linux device](#). Later in this tutorial, we'll add an NVIDIA module to our IoT Edge device.

Create a GPU-optimized virtual machine

To create a GPU-optimized virtual machine (VM), choosing the right size is important. Not all VM sizes accommodate GPU processing. In addition, there are different VM sizes for different workloads. For more information, see [GPU optimized virtual machine sizes](#) or try the [Virtual machines selector](#).

Let's create an IoT Edge VM with the [Azure Resource Manager \(ARM\)](#) template in GitHub, then configure it to be GPU-optimized.

1. Go to the IoT Edge VM deployment template in GitHub: [Azure/iotedge-vm-deploy](#).
2. Select the **Deploy to Azure** button, which initiates the creation of a custom VM for you in the Azure portal.
3. Fill out the **Custom deployment** fields with your Azure credentials and resources:

[+] [Expand table](#)

Property	Description or sample value
Subscription	Choose your Azure account subscription.
Resource group	Add your Azure resource group.
Region	<code>East US</code> GPU VMs aren't available in all regions.
Dns Label Prefix	Create a name for your VM.
Admin Username	<code>adminUser</code> Alternatively, create your own user name.
Device Connection String	Copy your connection string from your IoT Edge device, then paste here.
VM size	<code>Standard_NV6</code>
Authentication type	Choose either password or SSH Public Key , then create a password or key pair name if needed.

💡 Tip

Check which GPU VMs are supported in each region: [Products available by region](#).

To check [which region your Azure subscription allows](#), try this Azure command from the Azure portal. The `N` in `Standard_N` means it's a GPU-enabled VM.

Azure CLI

```
az vm list-skus --location <YOUR-REGION> --size Standard_N --all --output table
```

4. Select the **Review + create** button at the bottom, then the **Create** button.

Deployment can take up one minute to complete.

Install the NVIDIA extension

Now that we have a GPU-optimized VM, let's install the [NVIDIA extension](#) on the VM using the Azure portal.

1. Open your VM in the Azure portal and select **Extensions + applications** from the left menu.
2. Select **Add** and choose the **NVIDIA GPU Driver Extension** from the list, then select **Next**.
3. Choose **Review + create**, then **Create**. The deployment can take up to 30 minutes to complete.
4. To confirm the installation in the Azure portal, return to **Extensions + applications** menu in your VM. The new extension named `NvidiaGpuDriverLinux` should be in your extensions list and show **Provisioning succeeded** under **Status**.
5. To confirm the installation using Azure Cloud Shell, run this command to list your extensions. Replace the `<>` placeholders with your values:

Azure CLI

```
az vm extension list --resource-group <YOUR-RESOURCE-GROUP> --vm-name <YOUR-VM-NAME> -o table
```

6. With an NVIDIA module, we'll use the [NVIDIA System Management Interface program](#), also known as `nvidia-smi`.

From your device, install the `nvidia-smi` package based on your version of Ubuntu. For this tutorial, we'll install `nvidia-utils-515` for Ubuntu 20.04. Select `Y` when prompted in the installation.

Bash

```
sudo apt install nvidia-utils-515
```

Here's a list of all `nvidia-smi` versions. If you run `nvidia-smi` without installing it first, this list will print in your console.

```
sudo apt install nvidia-340          # version 340.108-0ubuntu5.20.04.2, or
sudo apt install nvidia-utils-390      # version 390.151-0ubuntu0.20.04.1
sudo apt install nvidia-utils-450-server # version 450.191.01-0ubuntu0.20.04.1
sudo apt install nvidia-utils-470      # version 470.129.06-0ubuntu0.20.04.1
sudo apt install nvidia-utils-470-server # version 470.129.06-0ubuntu0.20.04.1
sudo apt install nvidia-utils-510      # version 510.73.05-0ubuntu0.20.04.1
sudo apt install nvidia-utils-510-server # version 510.73.08-0ubuntu0.20.04.1
sudo apt install nvidia-utils-515      # version 515.48.07-0ubuntu0.20.04.2
sudo apt install nvidia-utils-515-server # version 515.48.07-0ubuntu0.20.04.1
sudo apt install nvidia-utils-435      # version 435.21-0ubuntu7
sudo apt install nvidia-utils-440      # version 440.82+really.440.64-0ubuntu6
sudo apt install nvidia-utils-418-server # version 418.226.00-0ubuntu0.20.04.2
```

7. After installation, run this command to confirm it's installed:

Bash

```
nvidia-smi
```

A confirmation table will appear, similar to this table.

```
Fri Jul 22 00:56:52 2022
+-----+
| NVIDIA-SMI 510.73.08     Driver Version: 510.73.08     CUDA Version: 11.6 |
+-----+
| GPU  Name      Persistence-M | Bus-Id      Disp.A  | Volatile Uncorr. ECC | |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
|                               |             |            | MIG M.               |
+-----+
| 0  Tesla M60        On           00000001:00:00.0 Off |                  Off | | |
| N/A   32C    P8    14W / 150W |             0MiB /  8192MiB |      0%     Default |
|                               |             |            | N/A                 |
+-----+
+-----+
| Processes:
| GPU  GI  CI      PID  Type  Process name                    GPU Memory |
| ID   ID              |          |                   Usage
+-----+
| No running processes found
+-----+
```

⚠ Note

The NVIDIA extension is a simplified way to install the NVIDIA drivers, but you may need more customization. For more information about custom installations on N-series VMs, see [Install NVIDIA GPU drivers on N-series VMs running Linux](#).

Enable a module with GPU acceleration

There are different ways to enable an IoT Edge module so that it uses a GPU for processing. One way is to configure an existing IoT Edge module on your device to become GPU-accelerated. Another way is to use a prefabricated container module, for example, a module from [NVIDIA DIGITS](#) that's already GPU-optimized. Let's see how both ways are done.

Enable GPU in an existing module using DeviceRequests

If you have an existing module on your IoT Edge device, adding a configuration using `DeviceRequests` in `createOptions` of the deployment manifest makes the module GPU-optimized. Follow these steps to configure an existing module.

1. Go to your IoT Hub in the Azure portal and choose **Devices** under the **Device management** menu.
2. Select your IoT Edge device to open it.
3. Select the **Set modules** tab at the top.
4. Select the module you want to enable for GPU use in the **IoT Edge Modules** list.
5. A side panel opens, choose the **Container Create Options** tab.
6. Copy this `HostConfig` JSON string and paste into the **Create options** box.

JSON

```
{  
  "HostConfig": {  
    "DeviceRequests": [  
      {  
        "Count": -1,  
        "Capabilities": [  
          "gpu"
```

```
        ]
      ]
    }
}
```

7. Select **Update**.

8. Select **Review + create**. The new `HostConfig` object is now visible in the `settings` of your module.

9. Select **Create**.

10. To confirm the new configuration works, run this command in your VM:

Bash

```
sudo docker inspect <YOUR-MODULE-NAME>
```

You should see the parameters you specified for `DeviceRequests` in the JSON printout in the console.

 **Note**

To understand the `DeviceRequests` parameter better, view the source code:
[moby/host_config.go ↗](#)

Enable a GPU in a prefabricated NVIDIA module

Let's add an [NVIDIA DIGITS ↗](#) module to the IoT Edge device and then allocate a GPU to the module by setting its environment variables. This NVIDIA module is already in a Docker container.

1. Select your IoT Edge device in the Azure portal from your IoT Hub's **Devices** menu.
2. Select the **Set modules** tab at the top.
3. Select **+ Add** under the IoT Edge modules heading and choose **IoT Edge Module**.
4. Provide a name in the **IoT Edge Module Name** field.
5. Under the **Module Settings** tab, add `nvidia/digits:6.0` to the **Image URI** field.
6. Select the **Environment Variables** tab.

7. Add the environment variable name `NVIDIA_VISIBLE_DEVICES` with the value `0`. This variable controls which GPUs are visible to the containerized application running on the edge device. The `NVIDIA_VISIBLE_DEVICES` environment variable can be set to a comma-separated list of device IDs, which correspond to the physical GPUs in the system. For example, if there are two GPUs in the system with device IDs 0 and 1, the variable can be set to "NVIDIA_VISIBLE_DEVICES=0,1" to make both GPUs visible to the container. In this article, since the VM only has one GPU, we will use the first (and only) one.

[+] Expand table

Name	Type	Value
NVIDIA_VISIBLE_DEVICES	Text	0

8. Select **Add**.

9. Select **Review + create**. Your deployment manifest properties will appear.

10. Select **Create** to create the module.

11. Select **Refresh** to update your module list. The module will take a couple of minutes to show *running* in the **Runtime status**, so keep refreshing the device.

12. From your device, run this command to confirm your new NVIDIA module exists and is running.

```
Bash
```

```
iotedge list
```

You should see your NVIDIA module in a list of modules on your IoT Edge device with a status of `running`.

NAME	STATUS	DESCRIPTION	Config
SimulatedTemperatureSensor	stopped	Stopped	mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.0
edgeAgent	running	Up an hour	mcr.microsoft.com/azureiotedge-agent:1.1
edgeHub	running	Up an hour	mcr.microsoft.com/azureiotedge-hub:1.1
nvidia-module	running	Up 13 minutes	nvidia/digits:6.0

ⓘ Note

For more information on the **NVIDIA DIGITS** container module, see the [Deep Learning Digits Documentation](#).

Clean up resources

If you want to continue with other IoT Edge tutorials, you can use the device that you created for this tutorial. Otherwise, you can delete the Azure resources that you created to avoid charges.

If you created your virtual machine and IoT hub in a new resource group, you can delete that group, which will delete all the associated resources. Double check the contents of the resource group to make sure that there's nothing you want to keep. If you don't want to delete the whole group, you can delete individual resources (virtual machine, device, or GPU-module) instead.

 **Important**

Deleting a resource group is irreversible.

Use the following command to remove your Azure resource group. It might take a few minutes to delete a resource group.

Azure CLI

```
az group delete --name <YOUR-RESOURCE-GROUP> --yes
```

You can confirm the resource group is removed by viewing the list of resource groups.

Azure CLI

```
az group list
```

Next steps

This article helped you set up your virtual machine and IoT Edge device to be GPU-accelerated. To run an application with a similar setup, try the learning path for [NVIDIA DeepStream development with Microsoft Azure](#). The Learn tutorial shows you how to develop optimized Intelligent Video Applications that can consume multiple video, image, and audio sources.

Tutorial: Monitor IoT Edge devices

Article • 06/13/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Use Azure Monitor workbooks to monitor the health and performance of your Azure IoT Edge deployments.

In this tutorial, you learn how to:

- ✓ Understand what metrics are shared by IoT Edge devices and how the metrics collector module handles them.
- ✓ Deploy the metrics collector module to an IoT Edge device.
- ✓ View curated visualizations of the metrics collected from the device.

Prerequisites

An IoT Edge device with the simulated temperature sensor module deployed to it. If you don't have a device ready, follow the steps in [Deploy your first IoT Edge module to a virtual Linux device](#) to create one using a virtual machine.

Understand IoT Edge metrics

Every IoT Edge device relies on two modules, the *runtime modules*, which manage the lifecycle and communication of all the other modules on a device. These modules are called the **IoT Edge agent** and the **IoT Edge hub**. To learn more about these modules, see [Understand the Azure IoT Edge runtime and its architecture](#).

Both of the runtime modules create metrics that allow you to remotely monitor how an IoT Edge device or its individual modules are performing. The IoT Edge agent reports on the state of individual modules and the host device, so creates metrics like how long a module has been running correctly, or the amount of RAM and percent of CPU being used on the device. The IoT Edge hub reports on communications on the device, so creates metrics like the total number of messages sent and received, or the time it takes

to resolve a direct method. For the full list of available metrics, see [Access built-in metrics](#).

These metrics are exposed automatically by both modules so that you can create your own solutions to access and report on these metrics. To make this process easier, Microsoft provides the [azureiotedge-metrics-collector module](#) that handles this process for those who don't have or want a custom solution. The metrics collector module collects metrics from the two runtime modules and any other modules you may want to monitor, and transports them off-device.

The metrics collector module works one of two ways to send your metrics to the cloud. The first option, which we'll use in this tutorial, is to send the metrics directly to Log Analytics. The second option, which is only recommended if your networking policies require it, is to send the metrics through IoT Hub and then set up a route to pass the metric messages to Log Analytics. Either way, once the metrics are in your Log Analytics workspace, they are available to view through Azure Monitor workbooks.

Create a Log Analytics workspace

A Log Analytics workspace is necessary to collect the metrics data and provides a query language and integration with Azure Monitor to enable you to monitor your devices.

1. Sign in to the [Azure portal](#).
2. Search for and select **Log Analytics workspaces**.
3. Select **Create** and then follow the prompts to create a new workspace.
4. Once your workspace is created, select **Go to resource**.
5. From the main menu under **Settings**, select **Agents management**.
6. Copy the values of **Workspace ID** and **Primary key**. You'll use these two values later in the tutorial to configure the metrics collector module to send the metrics to this workspace.

Retrieve your IoT hub resource ID

When you configure the metrics collector module, you give it the Azure Resource Manager resource ID for your IoT hub. Retrieve that ID now.

1. From the Azure portal, navigate to your IoT hub.
2. From the menu on the left, under **Settings**, select **Properties**.

3. Copy the value of **Resource ID**. It should have the format

```
/subscriptions/<subscription_id>/resourceGroups/<resource_group_name>/providers/Microsoft.Devices/IoTHubs/<iot_hub_name>.
```

Deploy the metrics collector module

Deploy the metrics collector module to every device that you want to monitor. It runs on the device like any other module, and watches its assigned endpoints for metrics to collect and send to the cloud.

Follow these steps to deploy and configure the collector module:

1. Sign in to the [Azure portal](#) and go to your IoT hub.
2. From the menu on the left, select **Devices** under the **Device management** menu.
3. Select the device ID of the target device from the list of IoT Edge devices to open the device details page.
4. On the upper menu bar, select **Set Modules**.
5. The first step of deploying modules from the portal is to declare which **Modules** should be on a device. If you are using the same device that you created in the quickstart, you should already see **SimulatedTemperatureSensor** listed. If not, add it now:
 - a. In the **IoT Edge modules** section, select **Add** then choose **IoT Edge Module**.
 - b. Update the following module settings:

[] Expand table

Setting	Value
IoT Module name	SimulatedTemperatureSensor
Image URI	mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:latest
Restart policy	always
Desired status	running

- c. Select **Next: Routes** to continue to configure routes.

- d. Add a route that sends all messages from the simulated temperature module to IoT Hub.

[\[+\] Expand table](#)

Setting	Value
Name	SimulatedTemperatureSensorToIoTHub
Value	FROM /messages/modules/SimulatedTemperatureSensor/* INTO \$upstream

6. Add and configure the metrics collector module:

- a. Select **Add** then choose **IoT Edge Module**.
- b. Search for and select **IoT Edge Metrics Collector**.
- c. Update the following module settings:

[\[+\] Expand table](#)

Setting	Value
IoT Module name	IoTEdgeMetricsCollector
Image URI	mcr.microsoft.com/azureiotedge-metrics-collector:latest
Restart policy	always
Desired status	running

If you want to use a different version or architecture of the metrics collector module, find the available images in the [Microsoft Artifact Registry](#).

- a. Navigate to the **Environment Variables** tab.
- b. Add the following text type environment variables:

[\[+\] Expand table](#)

Name	Value
ResourceId	Your IoT hub resource ID that you retrieved in a previous section.
UploadTarget	AzureMonitor

Name	Value
LogAnalyticsWorkspaceId	Your Log Analytics workspace ID that you retrieved in a previous section.
LogAnalyticsSharedKey	Your Log Analytics key that you retrieved in a previous section.

For more information about environment variable settings, see [Metrics collector configuration](#).

- c. Select **Apply** to save your changes.

 **Note**

If you want the collector module to send the metrics through IoT Hub, you would add a route to upstream similar to `FROM /messages/modules/<FROM_MODULE_NAME>//* INTO $upstream`. However, in this tutorial we're sending the metrics directly to Log Analytics. Therefore, it's not needed.

7. Select **Review + create** to continue to the final step for deploying modules.

8. Select **Create** to finish the deployment.

After completing the module deployment, you return to the device details page where you can see four modules listed as **Specified in Deployment**. It may take a few moments for all four modules to be listed as **Reported by Device**, which means that they've been successfully started and reported their status to IoT Hub. Refresh the page to see the latest status.

Monitor device health

It may take up to fifteen minutes for your device monitoring workbooks to be ready to view. Once you deploy the metrics collector module, it starts sending metrics messages to Log Analytics where they're organized within a table. The IoT Hub resource ID that you provided links the metrics that are ingested to the hub that they belong to. As a result, the curated IoT Edge workbooks can retrieve metrics by querying against the metrics table using the resource ID.

Azure Monitor provides three default workbook templates for IoT:

- The **Fleet View** workbook shows the health of devices across multiple IoT resources. The view allows configuring thresholds for determining device health

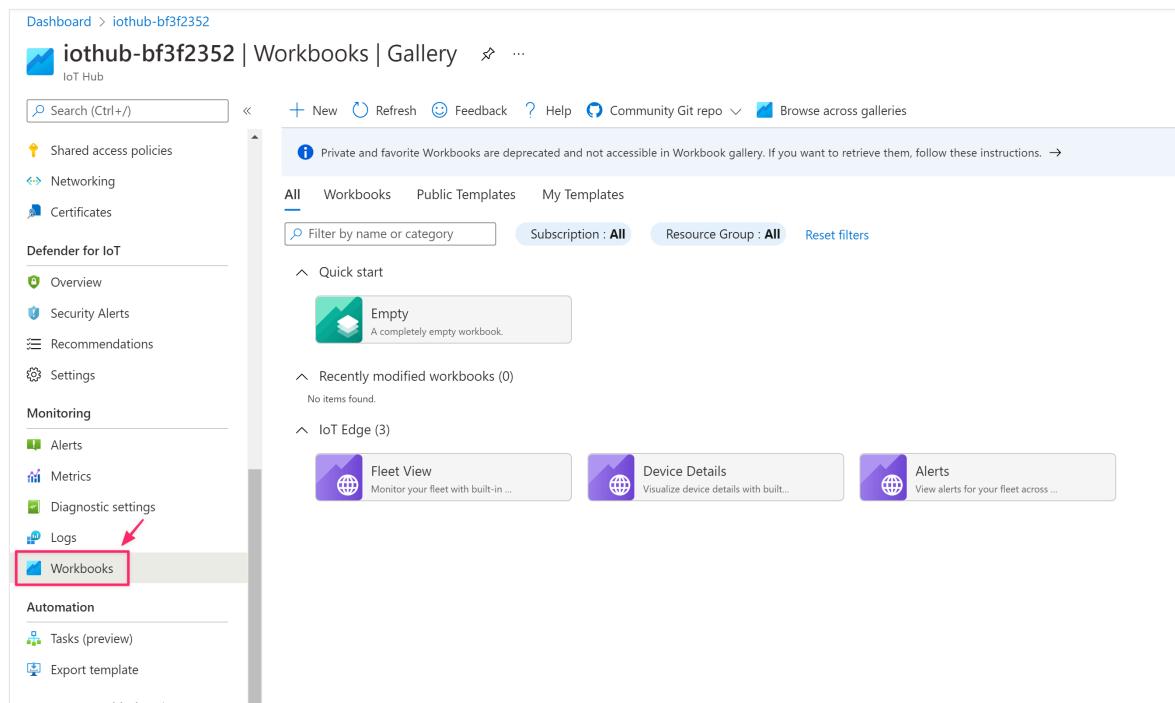
and presents aggregations of primary metrics, per-device.

- The **Device Details** workbook provides visualizations around three categories: messaging, modules, and host. The messaging view visualizes the message routes for a device and reports on the overall health of the messaging system. The modules view shows how the individual modules on a device are performing. The host view shows information about the host device including version information for host components and resource use.
- The **Alerts** workbook View presents alerts for devices across multiple IoT resources.

Explore the fleet view and health snapshot workbooks

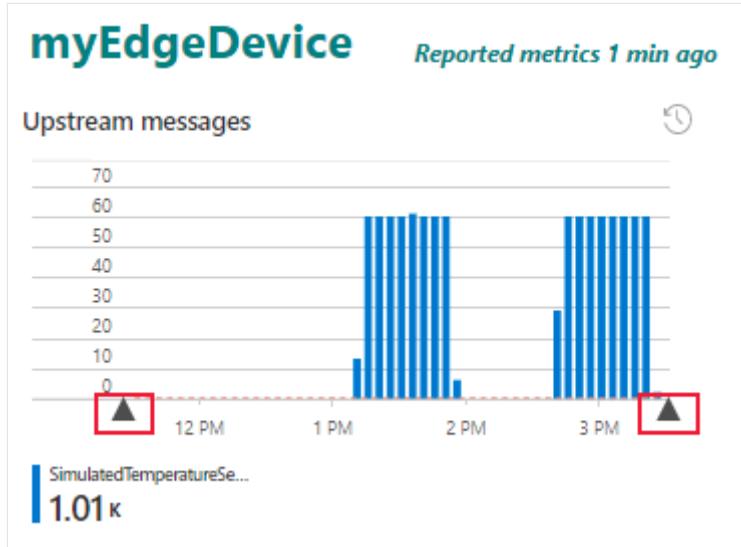
The fleet view workbook shows all of your devices, and lets you select specific devices to view their health snapshots. Use the following steps to explore the workbook visualizations:

1. Return to your IoT hub page in the Azure portal.
2. Scroll down in the main menu to find the **Monitoring** section, and select **Workbooks**.



3. Select the **Fleet View** workbook.
4. You should see your device that's running the metrics collector module. The device is listed as either **healthy** or **unhealthy**.
5. Select the device name to view detailed metrics from the device.

6. On any of the time charts, use the arrow icons under the X-axis or select the chart and drag your cursor to change the time range.



7. Close the health snapshot workbook. Select **Workbooks** from the fleet view workbook to return to the workbooks gallery.

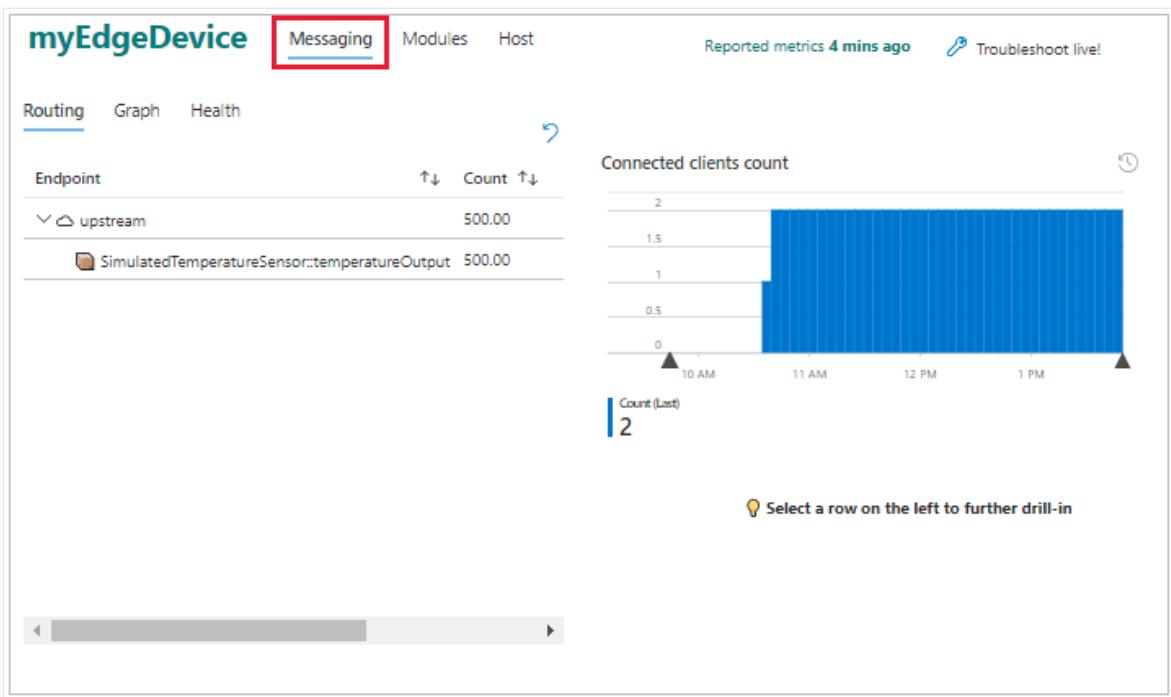
Explore the device details workbook

The device details workbook shows performance details for an individual device. Use the following steps to explore the workbook visualizations:

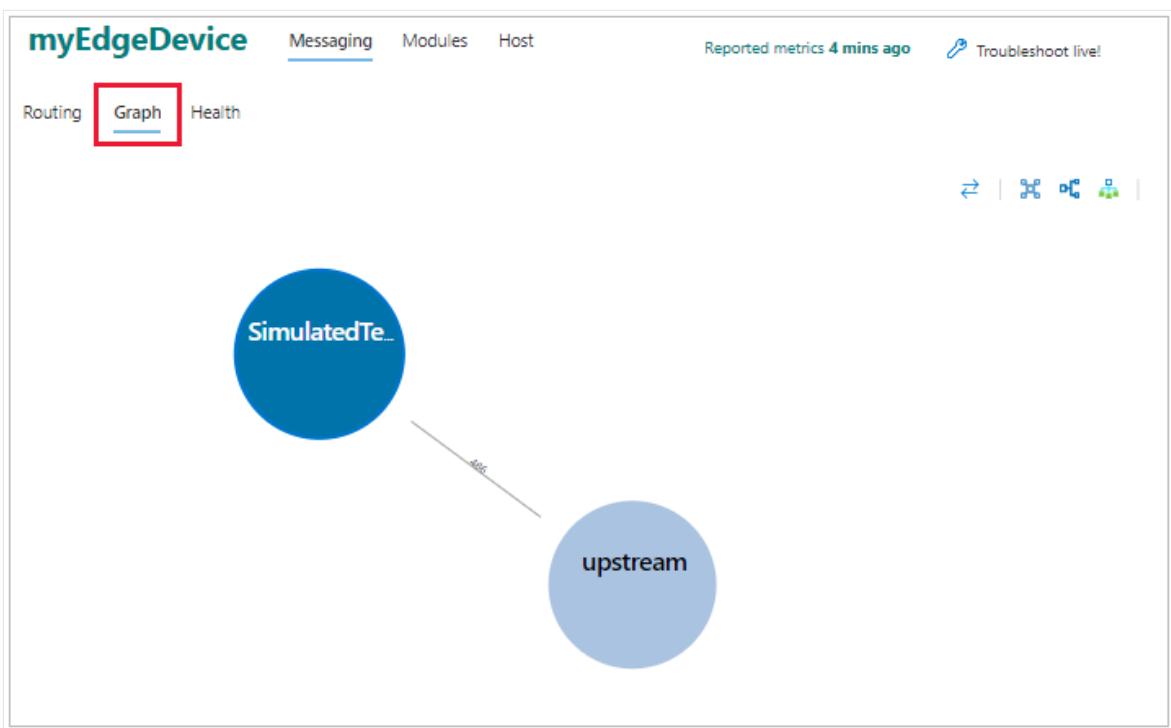
1. From the workbooks gallery, select the **IoT Edge device details** workbook.
2. The first page you see in the device details workbook is the **messaging** view with the **routing** tab selected.

On the left, a table displays the routes on the device, organized by endpoint. For our device, we see that the **upstream** endpoint, which is the special term used for routing to IoT Hub, is receiving messages from the **temperatureOutput** output of the simulated temperature sensor module.

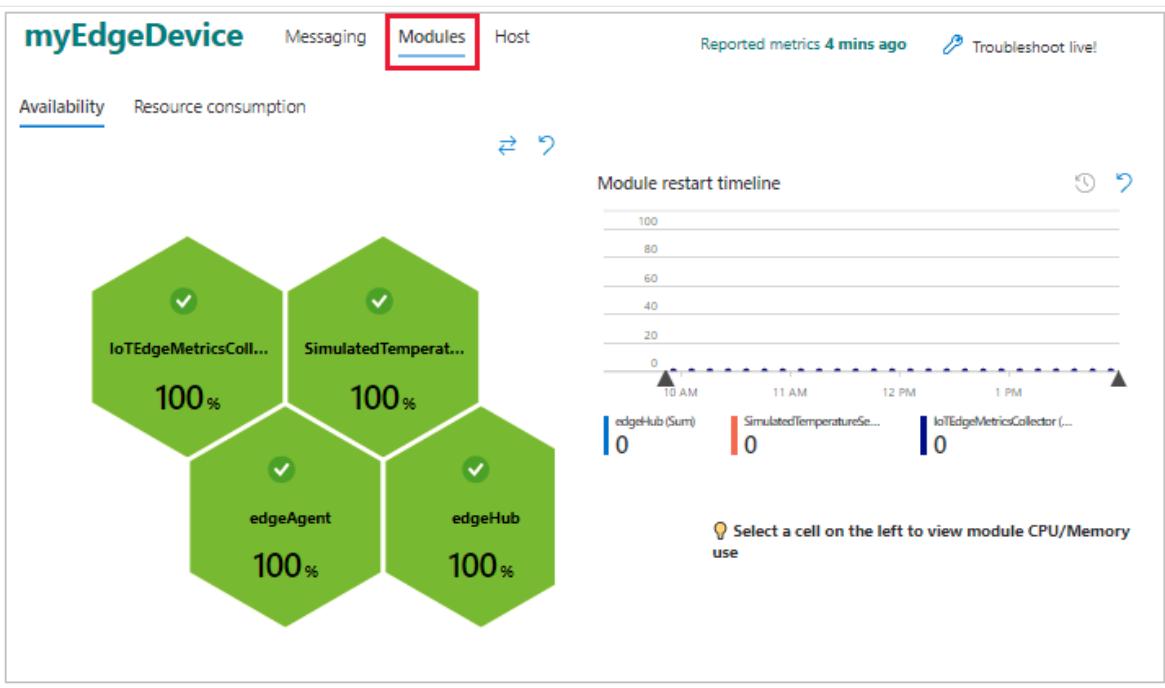
On the right, a graph keeps track of the number of connected clients over time. You can select and drag the graph to change the time range.



3. Select the **graph** tab to see a different visualization of the routes. On the graph page, you can drag and drop the different endpoints to rearrange the graph. This feature is helpful when you have many routes to visualize.



4. The **health** tab reports any issues with messaging, like dropped messages or disconnected clients.
5. Select the **modules** view to see the status of all the modules deployed on the device. You can select each of the modules to see details about how much CPU and memory they use.

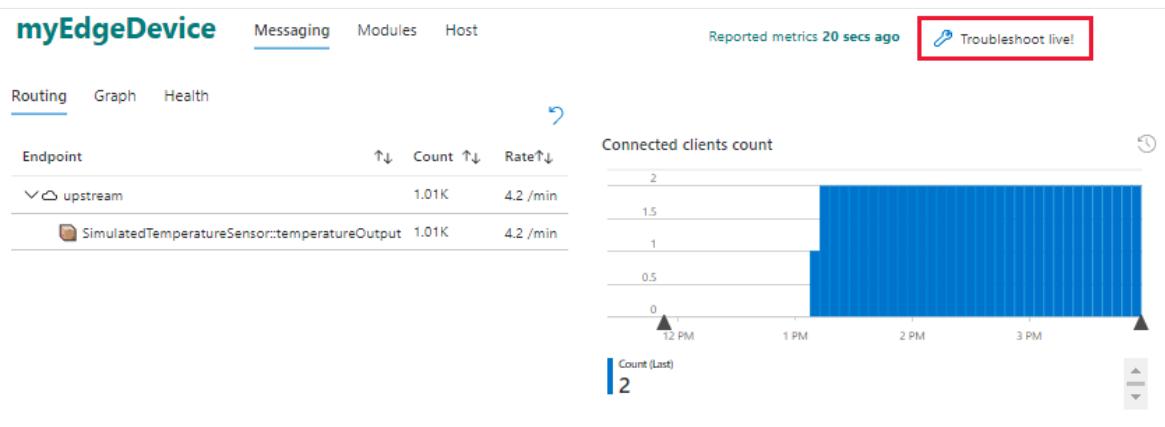


6. Select the **host** view to see information about the host device, including its operating system, the IoT Edge daemon version, and resource use.

View module logs

After viewing the metrics for a device, you might want to dive in further and inspect the individual modules. IoT Edge provides troubleshooting support in the Azure portal with a live module log feature.

1. From the device details workbook, select **Troubleshoot live!**.



2. The troubleshooting page opens to the **edgeAgent** logs from your IoT Edge device. If you selected a specific time range in the device details workbook, that setting is passed through to the troubleshooting page.
3. Use the dropdown menu to switch to the logs of other modules running on the device. Use the **Restart** button to restart a module.

The troubleshoot page can also be accessed from an IoT Edge device's details page. For more information, see [Troubleshoot IoT Edge devices from the Azure portal](#).

Next steps

As you continue through the rest of the tutorials, keep the metrics collector module on your devices and return to these workbooks to see how the information changes as you add more complex modules and routing.

Continue to the next tutorial where you set up your developer environment to start deploying custom modules to your devices.

[Develop Azure IoT Edge modules using Visual Studio Code](#)

Feedback

Was this page helpful?

[Yes](#)

[No](#)

[Provide product feedback ↗](#)

Understand the Azure IoT Edge runtime and its architecture

Article • 06/05/2024

Applies to: ✓ IoT Edge 1.5 ✓ IoT Edge 1.4

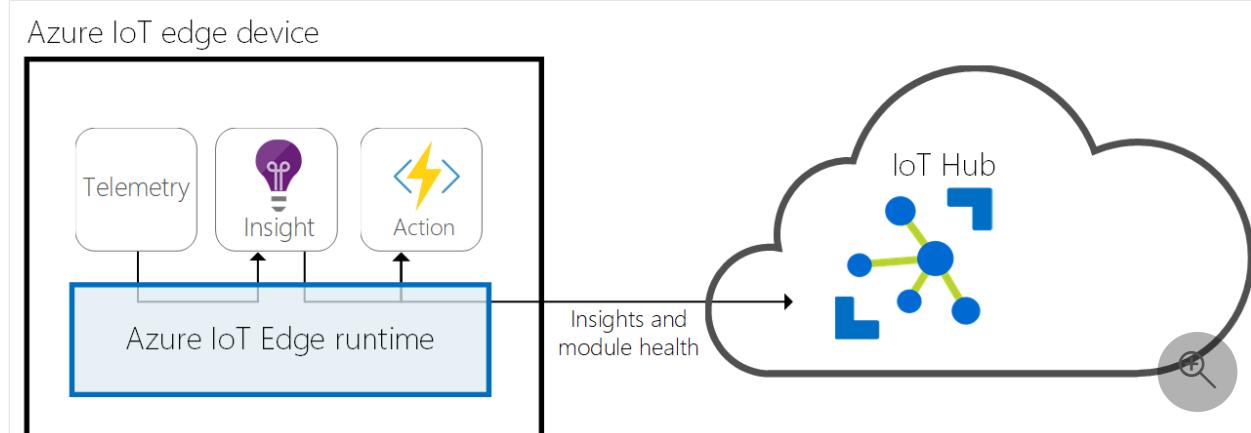
i Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The IoT Edge runtime is a collection of programs that turn a device into an IoT Edge device. Collectively, the IoT Edge runtime components enable IoT Edge devices to receive code to run at the edge and communicate the results.

The IoT Edge runtime is responsible for the following functions on IoT Edge devices:

- Install and update workloads on the device.
- Maintain Azure IoT Edge security standards on the device.
- Ensure that [IoT Edge modules](#) are always running.
- Report module health to the cloud for remote monitoring.
- Manage communication between:
 - Downstream devices and IoT Edge devices
 - Modules on an IoT Edge device
 - An IoT Edge device and the cloud
 - IoT Edge devices



The responsibilities of the IoT Edge runtime fall into two categories: communication and module management. These two roles are performed by two components that are part of the IoT Edge runtime. The *IoT Edge agent* deploys and monitors the modules, while the *IoT Edge hub* is responsible for communication.

Both the IoT Edge agent and the IoT Edge hub are modules, just like any other module running on an IoT Edge device. They're sometimes referred to as the *runtime modules*.

IoT Edge agent

The IoT Edge agent is one of two modules that make up the Azure IoT Edge runtime. It's responsible for instantiating modules, ensuring that they continue to run, and reporting the status of the modules back to IoT Hub. This configuration data is written as a property of the IoT Edge agent module twin.

The [IoT Edge security daemon](#) starts the IoT Edge agent on device startup. The agent retrieves its module twin from IoT Hub and inspects the deployment manifest. The deployment manifest is a JSON file that declares the modules that need to be started.

Each item in the deployment manifest contains specific information about a module and is used by the IoT Edge agent for controlling the module's lifecycle. For more information about all the properties used by the IoT Edge agent to control modules, read about the [Properties of the IoT Edge agent and IoT Edge hub module twins](#).

The IoT Edge agent sends runtime response to IoT Hub. Here's a list of possible responses:

- 200 - OK
- 400 - The deployment configuration is malformed or invalid.
- 417 - The device doesn't have a deployment configuration set.
- 412 - The schema version in the deployment configuration is invalid.
- 406 - The IoT Edge device is offline or not sending status reports.
- 500 - An error occurred in the IoT Edge runtime.

For more information about creating deployment manifests, see [Learn how to deploy modules and establish routes in IoT Edge](#).

Security

The IoT Edge agent plays a critical role in the security of an IoT Edge device. For example, it performs actions like verifying a module's image before starting it.

For more information about the Azure IoT Edge security framework, read about the [IoT Edge security manager](#).

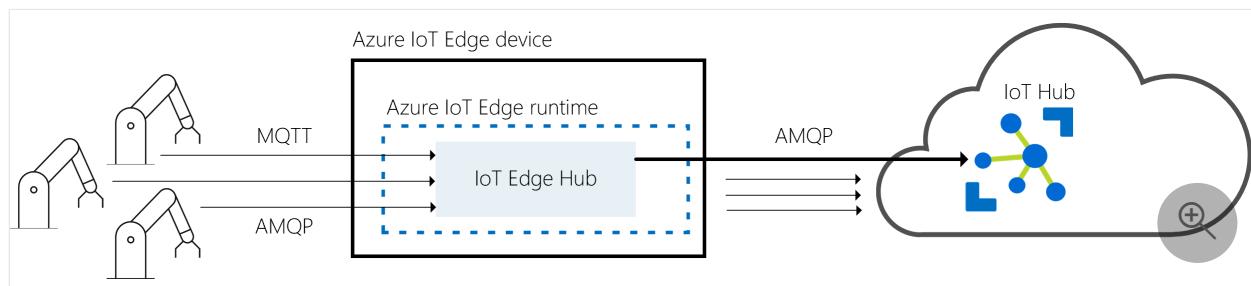
IoT Edge hub

The IoT Edge hub is the other module that makes up the Azure IoT Edge runtime. It acts as a local proxy for IoT Hub by exposing the same protocol endpoints as IoT Hub. This consistency means that clients can connect to the IoT Edge runtime just as they would to IoT Hub.

The IoT Edge hub isn't a full version of IoT Hub running locally. IoT Edge hub silently delegates some tasks to IoT Hub. For example, IoT Edge hub automatically downloads authorization information from IoT Hub on its first connection to enable a device to connect. After the first connection is established, authorization information is cached locally by IoT Edge hub. Future connections from that device are authorized without having to download authorization information from the cloud again.

Cloud communication

To reduce the bandwidth that your IoT Edge solution uses, the IoT Edge hub optimizes how many actual connections are made to the cloud. IoT Edge hub takes logical connections from modules or downstream devices and combines them for a single physical connection to the cloud. The details of this process are transparent to the rest of the solution. Clients think they have their own connection to the cloud even though they're all being sent over the same connection. The IoT Edge hub can either use the AMQP or the MQTT protocol to communicate upstream with the cloud, independently from protocols used by downstream devices. However, the IoT Edge hub currently only supports combining logical connections into a single physical connection by using AMQP as the upstream protocol and its multiplexing capabilities. AMQP is the default upstream protocol.



IoT Edge hub can determine whether it's connected to IoT Hub. If the connection is lost, IoT Edge hub saves messages or twin updates locally. Once a connection is reestablished, it syncs all the data. The location used for this temporary cache is

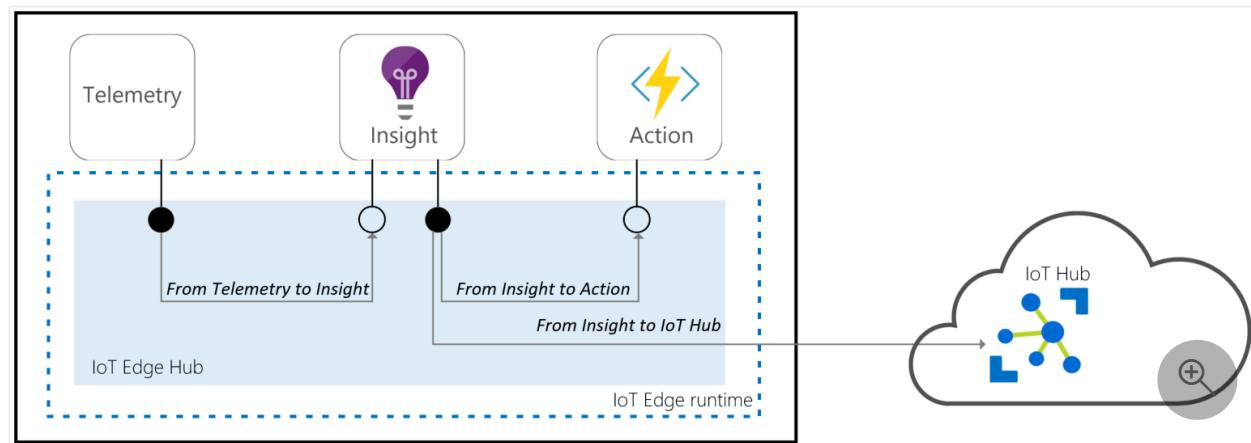
determined by a property of the IoT Edge hub's module twin. The size of the cache isn't capped and grows as long as the device has storage capacity. For more information, see [Offline capabilities](#).

Local communication

IoT Edge hub facilitates local communication. It enables device-to-module and module-to-module communications by brokering messages to keep devices and modules independent from each other. The IoT Edge hub supports the [message routing features supported by IoT Hub](#).

Using routing

The brokering mechanism uses the same routing features as IoT Hub to specify how messages are passed between devices or modules. First devices or modules specify the inputs on which they accept messages and the outputs to which they write messages. Then a solution developer can route messages between a source (for example, outputs), and a destination (for example, inputs), with potential filters.



Routing can be used by devices or modules built with the Azure IoT Device SDKs using the AMQP protocol. All messaging IoT Hub primitives (for example, telemetry), direct methods, C2D, twins, are supported but communication over user-defined topics isn't supported.

For more information about routes, see [Learn how to deploy modules and establish routes in IoT Edge](#).

Brokering mechanism features available:

[] [Expand table](#)

Features	Routing
D2C telemetry	✓
Local telemetry	✓
DirectMethods	✓
Twin	✓
C2D for devices	✓
Ordering	✓
Filtering	✓
User-defined topics	
Device-to-Device	
Local broadcasting	

Connecting to the IoT Edge hub

The IoT Edge hub accepts connections from device or module clients, either over the MQTT protocol or the AMQP protocol.

 **Note**

IoT Edge hub supports clients that connect using MQTT or AMQP. It does not support clients that use HTTP.

When a client connects to the IoT Edge hub, the following happens:

1. If Transport Layer Security (TLS) is used (recommended), a TLS channel is built to establish an encrypted communication between the client and the IoT Edge hub.
2. Authentication information is sent from the client to IoT Edge hub to identify itself.
3. IoT Edge hub authorizes or rejects the connection based on its authorization policy.

Secure connections (TLS)

By default, the IoT Edge hub only accepts connections secured with Transport Layer Security (TLS), for example, encrypted connections that a third party can't decrypt.

If a client connects on port 8883 (MQTTS) or 5671 (AMQPS) to the IoT Edge hub, a TLS channel must be built. During the TLS handshake, the IoT Edge hub sends its certificate chain that the client needs to validate. In order to validate the certificate chain, the root certificate of the IoT Edge hub must be installed as a trusted certificate on the client. If the root certificate isn't trusted, the client library is rejected by the IoT Edge hub with a certificate verification error.

The steps to follow to install this root certificate of the broker on device clients are described in the [transparent gateway](#) and in the [prepare a downstream device](#) documentation. Modules can use the same root certificate as the IoT Edge hub by using the IoT Edge daemon API.

Authentication

The IoT Edge Hub only accepts connections from devices or modules that have an IoT Hub identity. For example, those that are registered in IoT Hub and have one of the three client authentication methods supported by IoT Hub to prove their identity: [Symmetric keys authentication](#), [X.509 self-signed authentication](#), [X.509 CA signed authentication](#). These IoT Hub identities can be verified locally by the IoT Edge hub so connections can still be made while offline.

IoT Edge modules currently only support symmetric key authentication.

Authorization

By verifying that a client belongs to its set of trusted clients defined in IoT Hub. The set of trusted clients is specified by setting up parent/child or device/module relationships in IoT Hub. When a module is created in IoT Edge, a trust relationship is automatically established between this module and its IoT Edge device. This is the only authorization model supported by the routing brokering mechanism.

Remote configuration

The IoT Edge hub is entirely controlled by the cloud. It gets its configuration from IoT Hub via its [module twin](#). The twin contains a desired property called routes that declares how messages are passed within a deployment. For more information on routes, see [declare routes](#).

Additionally, several configurations can be done by setting up [environment variables on the IoT Edge hub ↗](#).

Runtime quality telemetry

IoT Edge collects anonymous telemetry from the host runtime and system modules to improve product quality. This information is called runtime quality telemetry. The collected telemetry is periodically sent as device-to-cloud messages to IoT Hub from the IoT Edge agent. These messages don't appear in customer's regular telemetry and don't consume any message quota.

The IoT Edge agent and hub generate metrics that you can collect to understand device performance. A subset of these metrics is collected by the IoT Edge Agent as part of runtime quality telemetry. The metrics collected for runtime quality telemetry are labeled with the tag `ms_telemetry`. For information about all the available metrics, see [Access built-in metrics](#).

Any personally or organizationally identifiable information, such as device and module names, are removed before upload to ensure the anonymous nature of the runtime quality telemetry.

The IoT Edge agent collects the telemetry every hour and sends one message to IoT Hub every 24 hours.

If you wish to opt out of sending runtime telemetry from your devices, there are two ways to do so:

- Set the `SendRuntimeQualityTelemetry` environment variable to `false` for `edgeAgent`
- Uncheck the option in the Azure portal during deployment.

Next steps

- [Understand Azure IoT Edge modules](#)
- [Learn how to deploy modules and establish routes in IoT Edge](#)
- [Learn how to publish and subscribe with IoT Edge](#)
- [Learn about IoT Edge runtime metrics](#)

Properties of the IoT Edge agent and IoT Edge hub module twins

Article • 03/21/2023

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The IoT Edge agent and IoT Edge hub are two modules that make up the IoT Edge runtime. For more information about the responsibilities of each runtime module, see [Understand the Azure IoT Edge runtime and its architecture](#).

This article provides the desired properties and reported properties of the runtime module twins. For more information on how to deploy modules on IoT Edge devices, see [Learn how to deploy modules and establish routes in IoT Edge](#).

A module twin includes:

- **Desired properties.** The solution backend can set desired properties, and the module can read them. The module can also receive notifications of changes in the desired properties. Desired properties are used along with reported properties to synchronize module configuration or conditions.
- **Reported properties.** The module can set reported properties, and the solution backend can read and query them. Reported properties are used along with desired properties to synchronize module configuration or conditions.

EdgeAgent desired properties

The module twin for the IoT Edge agent is called `$edgeAgent` and coordinates the communications between the IoT Edge agent running on a device and IoT Hub. The desired properties are set when applying a deployment manifest on a specific device as part of a single-device or at-scale deployment.

 Expand table

Property	Description	Required
imagePullPolicy	When to pull the image in either <i>OnCreate</i> or <i>Never</i> (<i>Never</i> can be used if the image is already on the device)	Yes
restartPolicy	When the module should be restarted. Possible values are: <i>Never</i> : don't restart module if not running, <i>Always</i> : always restart module if not running, <i>On-Unhealthy</i> : restart module if unhealthy. Unhealthy is what Docker reports based on a health check, for example "Unhealthy - the container is not working correctly", <i>On-Failed</i> : restart if Failed.	Yes
runtime.type	Has to be <i>docker</i> .	Yes
runtime.settings.minDockerVersion	Set to the minimum Docker version required by this deployment manifest.	Yes
runtime.settings.loggingOptions	A stringified JSON containing the logging options for the IoT Edge agent container. Docker logging options ↗	No
runtime.settings.registryCredentials.{registryId}.username	The username of the container registry. For Azure Container Registry, the username is usually the registry name. Registry credentials are necessary for any private module images.	No
runtime.settings.registryCredentials.{registryId}.password	The password for the container registry.	No
runtime.settings.registryCredentials.{registryId}.address	The address of the container registry. For Azure Container Registry,	No

Property	Description	Required
	the address is usually <code>{registry name}.azurecr.io</code> .	
schemaVersion	Either <code>1.0</code> or <code>1.1</code> . Version <code>1.1</code> was introduced with IoT Edge version <code>1.0.10</code> , and is recommended.	Yes
status	Desired status of the module: <code>Running</code> or <code>Stopped</code> .	Required
systemModules.edgeAgent.type	Has to be <code>docker</code> .	Yes
systemModules.edgeAgent.startupOrder	An integer value for the location a module has in the startup order. A <code>0</code> is first and <code>max integer</code> (<code>4294967295</code>) is last. If a value isn't provided, the default is <code>max integer</code> .	No
systemModules.edgeAgent.settings.image	The URI of the image of the IoT Edge agent. Currently, the IoT Edge agent isn't able to update itself.	Yes
systemModules.edgeAgent.settings.createOptions	A stringified JSON containing the options for the creation of the IoT Edge agent container. Docker create options ↗	No
systemModules.edgeAgent.configuration.id	The ID of the deployment that deployed this module.	IoT Hub sets this property when the manifest is applied using a deployment. Not part of a deployment manifest.
systemModules.edgeHub.type	Has to be <code>docker</code> .	Yes
systemModules.edgeHub.status	Has to be <code>running</code> .	Yes
systemModules.edgeHub.restartPolicy	Has to be <code>always</code> .	Yes

Property	Description	Required
systemModules.edgeHub.startupOrder	An integer value for which spot a module has in the startup order. A <i>0</i> is first and <i>max integer</i> (4294967295) is last. If a value isn't provided, the default is <i>max integer</i> .	No
systemModules.edgeHub.settings.image	The URI of the image of the IoT Edge hub.	Yes
systemModules.edgeHub.settings.createOptions	A stringified JSON containing the options for the creation of the IoT Edge hub container. Docker create options ↗	No
systemModules.edgeHub.configuration.id	The ID of the deployment that deployed this module.	IoT Hub sets this property when the manifest is applied using a deployment. Not part of a deployment manifest.
modules.{moduleId}.version	A user-defined string representing the version of this module.	Yes
modules.{moduleId}.type	Has to be <i>docker</i> .	Yes
modules.{moduleId}.status	{ <i>running</i> <i>stopped</i> }	Yes
modules.{moduleId}.restartPolicy	{ <i>never</i> <i>always</i> }	Yes
modules.{moduleId}.startupOrder	An integer value for the location a module has in the startup order. A <i>0</i> is first and <i>max integer</i> (4294967295) is last. If a value isn't provided, the default is <i>max integer</i> .	No
modules.{moduleId}.imagePullPolicy	{ <i>on-create</i> <i>never</i> }	No
modules.{moduleId}.env	A list of environment variables to pass to the	No

Property	Description	Required
	module. Takes the format " <name>": {"value": " <value>"}.	
modules.{moduleId}.settings.image	The URI to the module image.	Yes
modules.{moduleId}.settings.createOptions	A stringified JSON containing the options for the creation of the module container. Docker create options ↗	No
modules.{moduleId}.configuration.id	The ID of the deployment that deployed this module.	IoT Hub sets this property when the manifest is applied using a deployment. Not part of a deployment manifest.
version	The current iteration that has version, commit, and build.	No

EdgeAgent reported properties

The IoT Edge agent reported properties include three main pieces of information:

1. The status of the application of the last-seen desired properties;
2. The status of the modules currently running on the device, as reported by the IoT Edge agent; and
3. A copy of the desired properties currently running on the device.

The copy of the current desired properties is useful to tell whether the device has applied the latest deployment or is still running a previous deployment manifest.

Note

The reported properties of the IoT Edge agent are useful as they can be queried with the [IoT Hub query language](#) to investigate the status of deployments at scale.

For more information on how to use the IoT Edge agent properties for status, see [Understand IoT Edge deployments for single devices or at scale](#).

The following table does not include the information that is copied from the desired properties.

[+] Expand table

Property	Description
lastDesiredStatus.code	This status code refers to the last desired properties seen by the IoT Edge agent. Allowed values: <code>200</code> Success, <code>400</code> Invalid configuration, <code>412</code> Invalid schema version, <code>417</code> Desired properties are empty, <code>500</code> Failed.
lastDesiredStatus.description	Text description of the status.
lastDesiredVersion	This integer refers to the last version of the desired properties processed by the IoT Edge agent.
runtime.platform.OS	Reporting the OS running on the device.
runtime.platform.architecture	Reporting the architecture of the CPU on the device.
schemaVersion	Schema version of reported properties.
systemModules.edgeAgent.runtimeStatus	The reported status of IoT Edge agent: <code>{running unhealthy}</code> .
systemModules.edgeAgent.statusDescription	Text description of the reported status of the IoT Edge agent.
systemModules.edgeAgent.exitCode	The exit code reported by the IoT Edge agent container if the container exits.
systemModules.edgeAgent.lastStartTimeUtc	Time when IoT Edge agent was last started.
systemModules.edgeAgent.lastExitTimeUtc	Time when IoT Edge agent last exited.
systemModules.edgeHub.runtimeStatus	Status of IoT Edge hub: <code>{ running stopped failed backoff unhealthy }</code> .
systemModules.edgeHub.statusDescription	Text description of the status of IoT Edge hub, if unhealthy.
systemModules.edgeHub.exitCode	Exit code reported by the IoT Edge hub container, if the container exits.

Property	Description
systemModules.edgeHub.lastStartTimeUtc	Time when IoT Edge hub was last started.
systemModules.edgeHub.lastExitTimeUtc	Time when IoT Edge hub was last exited.
systemModules.edgeHub.lastRestartTimeUtc	Time when IoT Edge hub was last restarted.
systemModules.edgeHub.restartCount	Number of times this module was restarted as part of the restart policy.
modules.{moduleId}.runtimeStatus	Status of the module: { <i>running</i> <i>stopped</i> <i>failed</i> <i>backoff</i> <i>unhealthy</i> }.
modules.{moduleId}.statusDescription	Text description of the status of the module, if unhealthy.
modules.{moduleId}.exitCode	The exit code reported by the module container, if the container exits.
modules.{moduleId}.lastStartTimeUtc	Time when the module was last started.
modules.{moduleId}.lastExitTimeUtc	Time when the module was last exited.
modules.{moduleId}.lastRestartTimeUtc	Time when the module was last restarted.
modules.{moduleId}.restartCount	Number of times this module was restarted as part of the restart policy.
version	Version of the image. Example: "version": { "version": "1.2.7", "build": "50979330", "commit": "d3ec971caa0af0fc39d2c1f91aef21e95bd0c03c" }.

EdgeHub desired properties

The module twin for the IoT Edge hub is called `$edgeHub` and coordinates the communications between the IoT Edge hub running on a device and IoT Hub. The desired properties are set when applying a deployment manifest on a specific device as part of a single-device or at-scale deployment.

[\[+\] Expand table](#)

Property	Description	Required in the deployment manifest
schemaVersion	Either 1.0 or 1.1. Version 1.1 was introduced with IoT Edge	Yes

Property	Description	Required in the deployment manifest
	version 1.0.10, and is recommended.	
routes.{routeName}	A string representing an IoT Edge hub route. For more information, see Declare routes .	The <code>routes</code> element can be present but empty.
storeAndForwardConfiguration.timeToLiveSecs	The device time in seconds that IoT Edge hub keeps messages if disconnected from routing endpoints, whether IoT Hub or a local module. This time persists over any power offs or restarts. For more information, see Offline capabilities .	Yes

EdgeHub reported properties

[+] [Expand table](#)

Property	Description
lastDesiredVersion	This integer refers to the last version of the desired properties processed by the IoT Edge hub.
lastDesiredStatus.code	The status code referring to last desired properties seen by the IoT Edge hub. Allowed values: <code>200</code> Success, <code>400</code> Invalid configuration, <code>500</code> Failed
lastDesiredStatus.description	Text description of the status.
clients	All clients connected to edgeHub with the status and last connected time. Example: "clients": { "device2/SimulatedTemperatureSensor": { "status": "Connected", "lastConnectedTimeUtc": "2022-11-17T21:49:16.4781564Z" } }.
clients.{device or moduleId}.status	The connectivity status of this device or module. Possible values <code>connected</code> <code>disconnected</code> . Only module identities can be in disconnected state. Downstream devices connecting to IoT Edge hub appear only when connected.

Property	Description
clients.{device or moduleId}.lastConnectTime	Last time the device or module connected.
clients.{device or moduleId}.lastDisconnectTime	Last time the device or module disconnected.
schemaVersion	Schema version of reported properties.
version	Version of the image. Example: "version": { "version": "1.2.7", "build": "50979330", "commit": "d3ec971caa0af0fc39d2c1f91aef21e95bd0c03c" }.

Next steps

To learn how to use these properties to build out deployment manifests, see [Understand how IoT Edge modules can be used, configured, and reused](#).

Understand Azure IoT Edge modules

Article • 12/15/2022

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge lets you deploy and manage business logic on the edge in the form of *modules*. Azure IoT Edge modules are the smallest unit of computation managed by IoT Edge, and can contain Azure services (such as Azure Stream Analytics) or your own solution-specific code. To understand how modules are developed, deployed, and maintained, consider the four conceptual elements of a module:

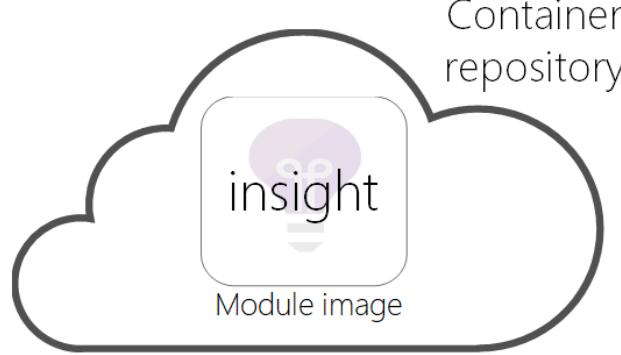
- A **module image** is a package containing the software that defines a module.
- A **module instance** is the specific unit of computation running the module image on an IoT Edge device. The module instance is started by the IoT Edge runtime.
- A **module identity** is a piece of information (including security credentials) stored in IoT Hub that is associated to each module instance.
- A **module twin** is a JSON document stored in IoT Hub that contains state information for a module instance, including metadata, configurations, and conditions.

Module images and instances

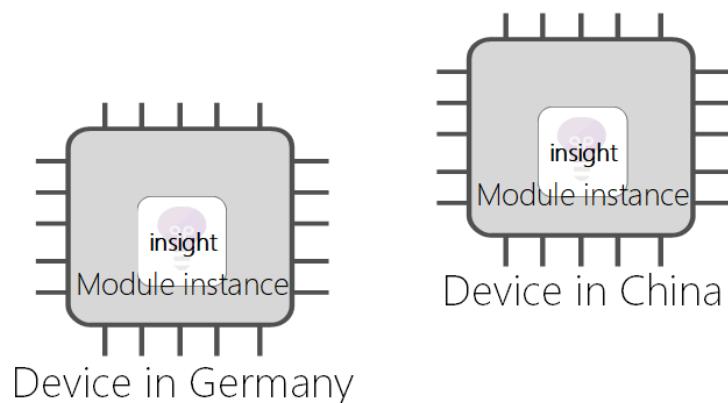
IoT Edge module images contain applications that take advantage of the management, security, and communication features of the IoT Edge runtime. You can develop your own module images, or export one from a supported Azure service, such as Azure Stream Analytics. The images exist in the cloud and they can be updated, changed, and deployed in different solutions. For instance, a module that uses machine learning to predict production line output exists as a separate image than a module that uses computer vision to control a drone.

Each time a module image is deployed to a device and started by the IoT Edge runtime, a new instance of that module is created. Two devices in different parts of the world could use the same module image. However, each device would have its own module instance when the module is started on the device.

Module image
lives in the cloud



Module instances
run on-premises



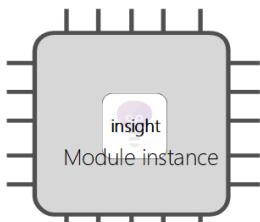
In implementation, module images exist as container images in a repository, and module instances are containers on devices.

Module identities

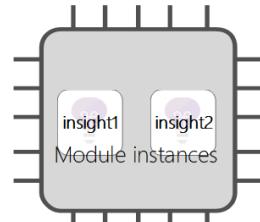
When a new module instance is created by the IoT Edge runtime, it gets a corresponding module identity. The module identity is stored in IoT Hub, and is used as the addressing and security scope for all local and cloud communications for that module instance.

The identity associated with a module instance depends on the identity of the device on which the instance is running and the name you provide to that module in your solution. For instance, if you call `insight` a module that uses an Azure Stream Analytics, and you deploy it on a device called `Hannover01`, the IoT Edge runtime creates a corresponding module identity called `/devices/Hannover01/modules/insight`.

Clearly, in scenarios when you need to deploy one module image multiple times on the same device, you can deploy the same image multiple times with different names.



Device in Germany



Device in China

Device /devices/Hannover01

Modules /devices/Hannover01/modules/insight

Device /devices/Shenzhen01

Modules /devices/Shenzhen01/modules/insight1
 /devices/Shenzhen01/modules/insight2

Module twins

Each module instance also has a corresponding module twin that you can use to configure the module instance. The instance and the twin are associated with each other through the module identity.

A module twin is a JSON document that stores module information and configuration properties. This concept parallels the [device twin](#) concept from IoT Hub. The structure of a module twin is the same as a device twin. The APIs used to interact with both types of twins are also the same. The only difference between the two is the identity used to instantiate the client SDK.

C#

```
// Create a ModuleClient object. This ModuleClient will act on behalf of a
// module since it is created with a module's connection string instead
// of a device connection string.
ModuleClient client = new ModuleClient.CreateFromEnvironmentAsync(settings);
await client.OpenAsync();

// Get the module twin
Twin twin = await client.GetTwinAsync();
```

Offline capabilities

Azure IoT Edge modules can operate offline indefinitely after syncing with IoT Hub at least once. IoT Edge devices can also extend this offline capability to other IoT devices. For more information, see [Understand extended offline capabilities for IoT Edge devices, modules, and downstream devices](#).

Next steps

- Understand the requirements and tools for developing IoT Edge modules
- Understand the Azure IoT Edge runtime and its architecture

How an IoT Edge device can be used as a gateway

Article • 06/04/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

IoT Edge devices can operate as gateways, providing a connection between other devices on the network and IoT Hub.

The IoT Edge hub module acts like IoT Hub, so it can handle connections from other devices that have an identity with the same IoT hub. This type of gateway pattern is called *transparent* because messages can pass from downstream devices to IoT Hub as though there were not a gateway between them.

For devices that don't or can't connect to IoT Hub on their own, IoT Edge gateways can provide that connection. This type of gateway pattern is called *translation* because the IoT Edge device has to perform processing on incoming downstream device messages before they can be forwarded to IoT Hub. These scenarios require additional modules on the IoT Edge gateway to handle the processing steps.

The transparent and translation gateway patterns are not mutually exclusive. A single IoT Edge device can function as both a transparent gateway and a translation gateway.

All gateway patterns provide the following benefits:

- **Analytics at the edge** - Use AI services locally to process data coming from downstream devices without sending full-fidelity telemetry to the cloud. Find and react to insights locally and only send a subset of data to IoT Hub.
- **Downstream device isolation** - The gateway device can shield all downstream devices from exposure to the internet. It can sit in between an operational technology (OT) network that does not have connectivity and an information technology (IT) network that provides access to the web. Similarly, devices that don't have the capability to connect to IoT Hub on their own can connect to a gateway device instead.

- **Connection multiplexing** - All devices connecting to IoT Hub through an IoT Edge gateway can use the same underlying connection. This multiplexing capability requires that the IoT Edge gateway uses AMQP as its upstream protocol.
- **Traffic smoothing** - The IoT Edge device will automatically implement exponential backoff if IoT Hub throttles traffic, while persisting the messages locally. This benefit makes your solution resilient to spikes in traffic.
- **Offline support** - The gateway device stores messages and twin updates that cannot be delivered to IoT Hub.

Transparent gateways

In the transparent gateway pattern, devices that theoretically could connect to IoT Hub can connect to a gateway device instead. The downstream devices have their own IoT Hub identities and connect using either MQTT or AMQP protocols. The gateway simply passes communications between the devices and IoT Hub. Both the devices and the users interacting with them through IoT Hub are unaware that a gateway is mediating their communications. This lack of awareness means the gateway is considered *transparent*.

For more information about how the IoT Edge hub manages communication between downstream devices and the cloud, see [Understand the Azure IoT Edge runtime and its architecture](#).

Beginning with version 1.2 of IoT Edge, transparent gateways can handle connections from downstream IoT Edge devices.

Parent and child relationships

You declare transparent gateway relationships in IoT Hub by setting the IoT Edge gateway as the *parent* of a downstream device *child* that connects to it.

Note

A downstream device emits data directly to the Internet or to gateway devices (IoT Edge-enabled or not). A child device can be a downstream device or a gateway device in a nested topology.

The parent/child relationship is established at three points in the gateway configuration:

Cloud identities

All devices in a transparent gateway scenario need cloud identities so they can authenticate to IoT Hub. When you create or update a device identity, you can set the device's parent or child devices. This configuration authorizes the parent gateway device to handle authentication for its child devices.

ⓘ Note

Setting the parent device in IoT Hub used to be an optional step for downstream devices that use symmetric key authentication. However, starting with version 1.1.0 every downstream device must be assigned to a parent device.

You can configure the IoT Edge hub to go back to the previous behavior by setting the environment variable **AuthenticationMode** to the value **CloudAndScope**.

Child devices can only have one parent. By default, a parent can have up to 100 children. You can change this limit by setting the **MaxConnectedClients** environment variable in the parent device's edgeHub module.

IoT Edge devices can be both parents and children in transparent gateway relationships. A hierarchy of multiple IoT Edge devices reporting to each other can be created. The top node of a gateway hierarchy can have up to five generations of children. For example, an IoT Edge device can have five layers of IoT Edge devices linked as children below it. But the IoT Edge device in the fifth generation cannot have any children, IoT Edge or otherwise.

Gateway discovery

A child device needs to be able to find its parent device on the local network. Configure gateway devices with a **hostname**, either a fully qualified domain name (FQDN) or an IP address, that its child devices use to locate it.

On downstream IoT devices, use the **gatewayHostname** parameter in the connection string to point to the parent device.

On downstream IoT Edge devices, use the **parent_hostname** parameter in the config file to point to the parent device.

Secure connection

Parent and child devices also need to authenticate their connections to each other. Each device needs a copy of a shared root CA certificate which the child devices use to verify that they are connecting to the proper gateway.

When multiple IoT Edge gateways connect to each other in a gateway hierarchy, all the devices in the hierarchy should use a single certificate chain.

Device capabilities behind transparent gateways

All IoT Hub primitives that work with IoT Edge's messaging pipeline also support transparent gateway scenarios. Each IoT Edge gateway has store and forward capabilities for messages coming through it.

Use the following table to see how different IoT Hub capabilities are supported for devices compared to devices behind gateways.

[Expand table](#)

Capability	IoT device	IoT behind a gateway	IoT Edge device	IoT Edge behind a gateway
Device-to-cloud (D2C) messages	✓	✓	✓	✓
Cloud-to-device (C2D) messages	✓	✓	✗	✗
Direct methods	✓	✓	✓	✓
Device twins and Module twins	✓	✓	✓	✓
File upload	✓	✗	✗	✗
Container image pulls			✓	✓
Blob upload			✓	✓

Container images can be downloaded, stored, and delivered from parent devices to child devices.

Blobs, including support bundles and logs, can be uploaded from child devices to parent devices.

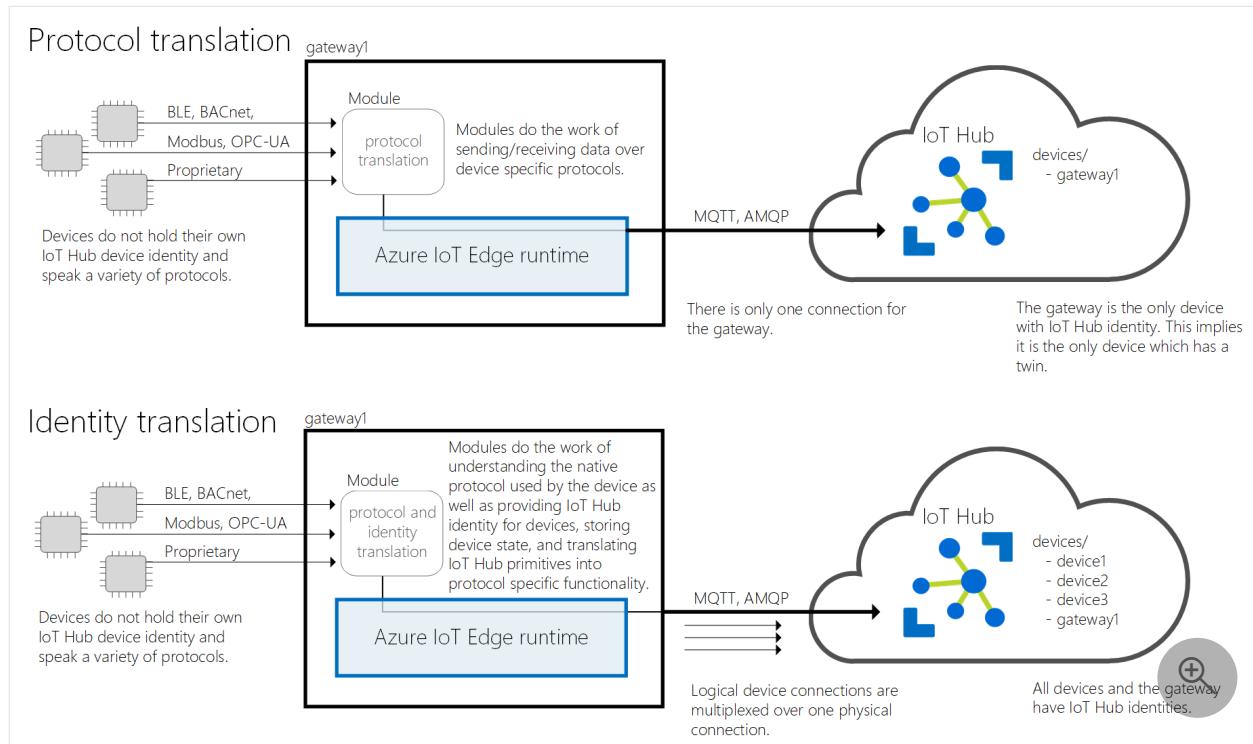
Translation gateways

If downstream devices can't connect to IoT Hub, then the IoT Edge gateway needs to act as a translator. Often, this pattern is required for devices that don't support MQTT,

AMQP, or HTTP. Since these devices can't connect to IoT Hub, they also can't connect to the IoT Edge hub module without some pre-processing.

Custom or third-party modules that are often specific to the downstream device's hardware or protocol need to be deployed to the IoT Edge gateway. These translation modules take the incoming messages and turn them into a format that can be understood by IoT Hub.

There are two patterns for translation gateways: *protocol translation* and *identity translation*.



Protocol translation

In the protocol translation gateway pattern, only the IoT Edge gateway has an identity with IoT Hub. The translation module receives messages from downstream devices, translates them into a supported protocol, and then the IoT Edge device sends the messages on behalf of the downstream devices. All information looks like it is coming from one device, the gateway. Downstream devices must embed additional identifying information in their messages if cloud applications want to analyze the data on a per-device basis. Additionally, IoT Hub primitives like twins and direct methods are only supported for the gateway device, not downstream devices. Gateways in this pattern are considered *opaque* in contrast to transparent gateways, because they obscure the identities of downstream devices.

Protocol translation supports devices that are resource constrained. Many existing devices are producing data that can power business insights; however they were not

designed with cloud connectivity in mind. Opaque gateways allow this data to be unlocked and used in an IoT solution.

Identity translation

The identity translation gateway pattern builds on protocol translation, but the IoT Edge gateway also provides an IoT Hub device identity on behalf of the downstream devices. The translation module is responsible for understanding the protocol used by the downstream devices, providing them identity, and translating their messages into IoT Hub primitives. Downstream devices appear in IoT Hub as first-class devices with twins and methods. A user can interact with the devices in IoT Hub and is unaware of the intermediate gateway device.

Identity translation provides the benefits of protocol translation and additionally allows for full manageability of downstream devices from the cloud. All devices in your IoT solution show up in IoT Hub regardless of the protocol they use.

Device capabilities behind translation gateways

The following table explains how IoT Hub features are extended to downstream devices in both translation gateway patterns.

[] Expand table

Capability	Protocol translation	Identity translation
Identities stored in the IoT Hub identity registry	Only the identity of the gateway device	Identities of all connected devices
Device twin	Only the gateway has a device and module twins	Each connected device has its own device twin
Direct methods and cloud-to-device messages	The cloud can only address the gateway device	The cloud can address each connected device individually
IoT Hub throttles and quotas	Apply to the gateway device	Apply to each device

When using the protocol translation pattern, all devices connecting through that gateway share the same cloud-to-device queue, which can contain at most 50 messages. Only use this pattern when few devices are connecting through each field gateway, and their cloud-to-device traffic is low.

The IoT Edge runtime does not include protocol or identity translation capabilities. These patterns require custom or third-party modules that are often specific to the hardware and protocol used. [Azure Marketplace](#) contains several protocol translation modules to choose from. For a sample that uses the identity translation pattern, see [Azure IoT Edge LoRaWAN Starter Kit](#).

Next steps

Learn the three steps to set up a transparent gateway:

- Configure an IoT Edge device to act as a transparent gateway
- Authenticate a downstream device to Azure IoT Hub
- Connect a downstream device to an Azure IoT Edge gateway

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Prepare to deploy your IoT Edge solution in production

Article • 08/07/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

When you're ready to take your IoT Edge solution from development into production, make sure that it's configured for ongoing performance.

The information provided in this article isn't all equal. To help you prioritize, each section starts with lists that divide the work into two sections: **important** to complete before going to production, or **helpful** for you to know.

Device configuration

IoT Edge devices can be anything from a Raspberry Pi to a laptop to a virtual machine running on a server. You may have access to the device either physically or through a virtual connection, or it may be isolated for extended periods of time. Either way, you want to make sure that it's configured to work appropriately.

- **Important**
 - Install production certificates
 - Have a device management plan
 - Use Moby as the container engine. If you are using Ubuntu Core snaps, the Docker snap is serviced by Canonical and supported for production scenarios.
- **Helpful**
 - Choose upstream protocol

Install production certificates

Every IoT Edge device in production needs a device certificate authority (CA) certificate installed on it. That CA certificate is then declared to the IoT Edge runtime in the config

file. For development and testing scenarios, the IoT Edge runtime creates temporary certificates if no certificates are declared in the config file. However, these temporary certificates expire after three months and aren't secure for production scenarios. For production scenarios, you should provide your own Edge CA certificate, either from a self-signed certificate authority or purchased from a commercial certificate authority.

To understand the role of the Edge CA certificate, see [How Azure IoT Edge uses certificates](#).

For more information about how to install certificates on an IoT Edge device and reference them from the config file, see [Manage certificate on an IoT Edge device](#).

Have a device management plan

Before you put any device in production you should know how you're going to manage future updates. For an IoT Edge device, the list of components to update may include:

- Device firmware
- Operating system libraries
- Container engine, like Moby
- IoT Edge
- CA certificates

[Device Update for IoT Hub](#) is a service that enables you to deploy over-the-air updates (OTA) for your IoT Edge devices.

Alternative methods for updating IoT Edge require physical or SSH access to the IoT Edge device. For more information, see [Update the IoT Edge runtime](#). To update multiple devices, consider adding the update steps to a script or use an automation tool like Ansible.

Container engine

A container engine is a prerequisite for any IoT Edge device. The moby-engine is supported in production. If you are using Ubuntu Core snaps, the Docker snap is serviced by Canonical and supported for production scenarios. Other container engines, like Docker, do work with IoT Edge and it's ok to use these engines for development. The moby-engine can be redistributed when used with Azure IoT Edge, and Microsoft provides servicing for this engine.

Choose upstream protocol

You can configure the protocol (which determines the port used) for upstream communication to IoT Hub for both the IoT Edge agent and the IoT Edge hub. The default protocol is AMQP, but you may want to change that depending on your network setup.

The two runtime modules both have an **UpstreamProtocol** environment variable. The valid values for the variable are:

- MQTT
- AMQP
- MQTTWS
- AMQPWS

Configure the UpstreamProtocol variable for the IoT Edge agent in the config file on the device itself. For example, if your IoT Edge device is behind a proxy server that blocks AMQP ports, you may need to configure the IoT Edge agent to use AMQP over WebSocket (AMQPWS) to establish the initial connection to IoT Hub.

Once your IoT Edge device connects, be sure to continue configuring the UpstreamProtocol variable for both runtime modules in future deployments. An example of this process is provided in [Configure an IoT Edge device to communicate through a proxy server](#).

Deployment

- Helpful
 - Be consistent with upstream protocol
 - Set up host storage for system modules
 - Reduce memory space used by the IoT Edge hub
 - Use correct module images in deployment manifests
 - Be mindful of twin size limits when using custom modules
 - Configure how updates to modules are applied

Be consistent with upstream protocol

If you configured the IoT Edge agent on your IoT Edge device to use a different protocol than the default AMQP, then you should declare the same protocol in all future deployments. For example, if your IoT Edge device is behind a proxy server that blocks AMQP ports, you probably configured the device to connect over AMQP over WebSocket (AMQPWS). When you deploy modules to the device, configure the same AMQPWS protocol for the IoT Edge agent and IoT Edge hub, or else the default AMQP overrides the settings and prevents you from connecting again.

You only have to configure the `UpstreamProtocol` environment variable for the IoT Edge agent and IoT Edge hub modules. Any additional modules adopt whatever protocol is set in the runtime modules.

An example of this process is provided in [Configure an IoT Edge device to communicate through a proxy server](#).

Set up host storage for system modules

The IoT Edge hub and agent modules use local storage to maintain state and enable messaging between modules, devices, and the cloud. For better reliability and performance, configure the system modules to use storage on the host filesystem.

For more information, see [Host storage for system modules](#).

Reduce memory space used by IoT Edge hub

If you're deploying constrained devices with limited memory available, you can configure IoT Edge hub to run in a more streamlined capacity and use less disk space. These configurations do limit the performance of the IoT Edge hub, however, so find the right balance that works for your solution.

Don't optimize for performance on constrained devices

The IoT Edge hub is optimized for performance by default, so it attempts to allocate large chunks of memory. This configuration can cause stability problems on smaller devices like the Raspberry Pi. If you're deploying devices with constrained resources, you may want to set the `OptimizeForPerformance` environment variable to `false` on the IoT Edge hub.

When `OptimizeForPerformance` is set to `true`, the MQTT protocol head uses the `PooledByteBufferAllocator`, which has better performance but allocates more memory. The allocator does not work well on 32-bit operating systems or on devices with low memory. Additionally, when optimized for performance, RocksDb allocates more memory for its role as the local storage provider.

For more information, see [Stability issues on smaller devices](#).

Disable unused protocols

Another way to optimize the performance of the IoT Edge hub and reduce its memory usage is to turn off the protocol heads for any protocols that you're not using in your

solution.

Protocol heads are configured by setting boolean environment variables for the IoT Edge hub module in your deployment manifests. The three variables are:

- `amqpSettings_enabled`
- `mqttSettings_enabled`
- `httpSettings_enabled`

All three variables have *two underscores* and can be set to either true or false.

Reduce storage time for messages

The IoT Edge hub module stores messages temporarily if they cannot be delivered to IoT Hub for any reason. You can configure how long the IoT Edge hub holds on to undelivered messages before letting them expire. If you have memory concerns on your device, you can lower the `timeToLiveSecs` value in the IoT Edge hub module twin.

The default value of the `timeToLiveSecs` parameter is 7200 seconds, which is two hours.

Use correct module images in deployment manifests

If an empty or wrong module image is used, the Edge agent retries to load the image, which causes extra traffic to be generated. Add the correct images to the deployment manifest to avoid generating unnecessary traffic.

Don't use debug versions of module images

When moving from test scenarios to production scenarios, remember to remove debug configurations from deployment manifests. Check that none of the module images in the deployment manifests have the `.debug` suffix. If you added create options to expose ports in the modules for debugging, remove those create options as well.

Be mindful of twin size limits when using custom modules

The deployment manifest that contains custom modules is part of the `EdgeAgent` twin. Review the [limitation on module twin size](#).

If you deploy a large number of modules, you might exhaust this twin size limit. Consider some common mitigations to this hard limit:

- Store any configuration in the custom module twin, which has its own limit.
- Store some configuration that points to a non-space-limited location (that is, to a blob store).

Configure how updates to modules are applied

When a deployment is updated, Edge Agent receives the new configuration as a twin update. If the new configuration has new or updated module images, by default, Edge Agent sequentially processes each module:

1. The updated image is downloaded
2. The running module is stopped
3. A new module instance is started
4. The next module update is processed

In some cases, for example when dependencies exist between modules, it may be desirable to first download all updated module images before restarting any running modules. This module update behavior can be configured by setting an IoT Edge Agent environment variable `ModuleUpdateMode` to string value `WaitForAllPulls`. For more information, see [IoT Edge Environment Variables](#).

```
JSON

"modulesContent": {
  "$edgeAgent": {
    "properties.desired": {
      ...
      "systemModules": {
        "edgeAgent": {
          "env": {
            "ModuleUpdateMode": {
              "value": "WaitForAllPulls"
            }
          ...
        }
      }
    }
  }
}
```

Container management

- **Important**
 - Use tags to manage versions
 - Manage volumes
- **Helpful**
 - Store runtime containers in your private registry
 - Configure image garbage collection

Use tags to manage versions

A tag is a docker concept that you can use to distinguish between versions of docker containers. Tags are suffixes like 1.5 that go on the end of a container repository. For example, [mcr.microsoft.com/azureiotedge-agent:1.5](https://mcr.microsoft.com.azureiotedge-agent:1.5). Tags are mutable and can be changed to point to another container at any time, so your team should agree on a convention to follow as you update your module images moving forward.

Tags also help you to enforce updates on your IoT Edge devices. When you push an updated version of a module to your container registry, increment the tag. Then, push a new deployment to your devices with the tag incremented. The container engine recognizes the incremented tag as a new version and pulls the latest module version down to your device.

Tags for the IoT Edge runtime

The IoT Edge agent and IoT Edge hub images are tagged with the IoT Edge version that they are associated with. There are two different ways to use tags with the runtime images:

- **Rolling tags** - Use only the first two values of the version number to get the latest image that matches those digits. For example, 1.5 is updated whenever there's a new release to point to the latest 1.5.x version. If the container runtime on your IoT Edge device pulls the image again, the runtime modules are updated to the latest version. Deployments from the Azure portal default to rolling tags. *This approach is suggested for development purposes.*
- **Specific tags** - Use all three values of the version number to explicitly set the image version. For example, 1.5.0 won't change after its initial release. You can declare a new version number in the deployment manifest when you're ready to update. *This approach is suggested for production purposes.*

Manage volumes

IoT Edge does not remove volumes attached to module containers. This behavior is by design, as it allows persisting the data across container instances such as upgrade scenarios. However, if these volumes are left unused, then it may lead to disk space exhaustion and subsequent system errors. If you use docker volumes in your scenario, then we encourage you to use docker tools such as [docker volume prune](#) and [docker volume rm](#) to remove the unused volumes, especially for production scenarios.

Store runtime containers in your private registry

You know how to store container images for custom code modules in your private Azure registry, but you can also use it to store public container images such as the **edgeAgent** and **edgeHub** runtime modules. Doing so may be required if you have tight firewall restrictions as these runtime containers are stored in the Microsoft Container Registry (MCR).

The following steps illustrate how to pull a Docker image of **edgeAgent** and **edgeHub** to your local machine, retag it, push it to your private registry, then update your configuration file so your devices know to pull the image from your private registry.

1. Pull the **edgeAgent** Docker image from the Microsoft registry. Update the version number if needed.

Bash

```
# Pull edgeAgent image
docker pull mcr.microsoft.com/azureiotedge-agent:1.5

# Pull edgeHub image
docker pull mcr.microsoft.com/azureiotedge-hub:1.5
```

2. List all your Docker images, find the **edgeAgent** and **edgeHub** images, then copy their image IDs.

Bash

```
docker images
```

3. Retag your **edgeAgent** and **edgeHub** images. Replace the values in brackets with your own.

Bash

```
# Retag your edgeAgent image
docker tag <my-image-id> <registry-name/server>/azureiotedge-agent:1.5

# Retag your edgeHub image
docker tag <my-image-id> <registry-name/server>/azureiotedge-hub:1.5
```

4. Push your **edgeAgent** and **edgeHub** images to your private registry. Replace the value in brackets with your own.

Bash

```
# Push your edgeAgent image to your private registry
docker push <registry-name/server>/azureiotedge-agent:1.5

# Push your edgeHub image to your private registry
docker push <registry-name/server>/azureiotedge-hub:1.5
```

5. Update the image references in the *deployment.template.json* file for the **edgeAgent** and **edgeHub** system modules, by replacing `mcr.microsoft.com` with your own "registry-name/server" for both modules.
6. Open a text editor on your IoT Edge device to change the configuration file so it knows about your private registry image.

Bash

```
sudo nano /etc/aziot/config.toml
```

7. In the text editor, change your image values under `[agent.config]`. Replace the values in brackets with your own.

toml

```
[agent.config]
image = "<registry-name/server>/azureiotedge-agent:1.5"
```

8. If your private registry requires authentication, set the authentication parameters in `[agent.config.auth]`.

toml

```
[agent.config.auth]
serveraddress = "<login-server>" # Almost always equivalent to
<registry-name/server>
username = "<username>"
password = "<password>"
```

9. Save your changes and exit your text editor.

10. Apply the IoT Edge configuration change.

Bash

```
sudo iotedge config apply
```

Your IoT Edge runtime restarts.

For more information, see:

- [Configure the IoT Edge agent](#)
- [Azure IoT Edge Agent ↗](#)
- [Azure IoT Edge Hub ↗](#)

Configure image garbage collection

Image garbage collection is a feature in IoT Edge v1.4 and later to automatically clean up Docker images that are no longer used by IoT Edge modules. It only deletes Docker images that were pulled by the IoT Edge runtime as part of a deployment. Deleting unused Docker images helps conserve disk space.

The feature is implemented in IoT Edge's host component, the `aziot-edged` service and enabled by default. Cleanup is done every day at midnight (device local time) and removes unused Docker images that were last used seven days ago. The parameters to control cleanup behavior are set in the `config.toml` and explained later in this section. If parameters aren't specified in the configuration file, the default values are applied.

For example, the following is the `config.toml` image garbage collection section using default values:

```
toml
[image_garbage_collection]
enabled = true
cleanup_recurrence = "1d"
image_age_cleanup_threshold = "7d"
cleanup_time = "00:00"
```

The following table describes image garbage collection parameters. All parameters are **optional** and can be set individually to change the default settings.

[+] Expand table

Parameter	Description	Required	Default value
<code>enabled</code>	Enables the image garbage collection. You may choose to disable the feature by changing this setting to <code>false</code> .	Optional	true

Parameter	Description	Required	Default value
<code>cleanup_recurrence</code>	<p>Controls the recurrence frequency of the cleanup task. Must be specified as a multiple of days and can't be less than one day.</p> <p>For example: 1d, 2d, 6d, etc.</p>	Optional	1d
<code>image_age_cleanup_threshold</code>	<p>Defines the minimum age threshold of unused images before considering for cleanup and must be specified in days. You can specify as <i>0d</i> to clean up the images as soon as they're removed from the deployment.</p> <p>Images are considered unused <i>after</i> they've been removed from the deployment.</p>	Optional	7d
<code>cleanup_time</code>	<p>Time of day, <i>in device local time</i>, when the cleanup task runs. Must be in 24-hour HH:MM format.</p>	Optional	00:00

Networking

- **Helpful**
 - Review outbound/inbound configuration
 - Allow connections from IoT Edge devices
 - Configure communication through a proxy
 - Set DNS server in container engine settings

Review outbound/inbound configuration

Communication channels between Azure IoT Hub and IoT Edge are always configured to be outbound. For most IoT Edge scenarios, only three connections are necessary. The container engine needs to connect with the container registry (or registries) that holds the module images. The IoT Edge runtime needs to connect with IoT Hub to retrieve device configuration information, and to send messages and telemetry. And if you use automatic provisioning, IoT Edge needs to connect to the Device Provisioning Service. For more information, see [Firewall and port configuration rules](#).

Allow connections from IoT Edge devices

If your networking setup requires that you explicitly permit connections made from IoT Edge devices, review the following list of IoT Edge components:

- **IoT Edge agent** opens a persistent AMQP/MQTT connection to IoT Hub, possibly over WebSockets.
- **IoT Edge hub** opens a single persistent AMQP connection or multiple MQTT connections to IoT Hub, possibly over WebSockets.
- **IoT Edge service** makes intermittent HTTPS calls to IoT Hub.

In all three cases, the fully qualified domain name (FQDN) would match the pattern `*.azure-devices.net`.

Container registries

The **Container engine** makes calls to container registries over HTTPS. To retrieve the IoT Edge runtime container images, the FQDN is `mcr.microsoft.com`. The container engine connects to other registries as configured in the deployment.

This checklist is a starting point for firewall rules:

 Expand table

FQDN (* = wildcard)	Outbound TCP Ports	Usage
<code>mcr.microsoft.com</code>	443	Microsoft Container Registry
<code>*.data.mcr.microsoft.com</code>	443	Data endpoint providing content delivery
<code>*.cdn.azurecr.io</code>	443	Deploy modules from the Marketplace to devices
<code>global.azure-devices-provisioning.net</code>	443	Device Provisioning Service access (optional)
<code>*.azurecr.io</code>	443	Personal and third-party container registries
<code>*.blob.core.windows.net</code>	443	Download Azure Container Registry image deltas from blob storage
<code>*.azure-devices.net</code>	5671, 8883, 443 ¹	IoT Hub access
<code>*.docker.io</code>	443	Docker Hub access (optional)

¹Open port 8883 for secure MQTT or port 5671 for secure AMQP. If you can only make connections via port 443 then either of these protocols can be run through a WebSocket

tunnel.

Since the IP address of an IoT hub can change without notice, always use the FQDN to allowlist configuration. To learn more, see [Understanding the IP address of your IoT Hub](#).

Some of these firewall rules are inherited from Azure Container Registry. For more information, see [Configure rules to access an Azure container registry behind a firewall](#).

You can enable dedicated data endpoints in your Azure Container registry to avoid wildcard allowlisting of the `*.blob.core.windows.net` FQDN. For more information, see [Enable dedicated data endpoints](#).

Note

To provide a consistent FQDN between the REST and data endpoints, beginning **June 15, 2020** the Microsoft Container Registry data endpoint will change from

`*.cdn.mscr.io` to `*.data.mcr.microsoft.com`

For more information, see [Microsoft Container Registry client firewall rules configuration](#)

If you don't want to configure your firewall to allow access to public container registries, you can store images in your private container registry, as described in [Store runtime containers in your private registry](#).

Azure IoT Identity Service

The [IoT Identity Service](#) provides provisioning and cryptographic services for Azure IoT devices. The identity service checks if the installed version is the latest version. The check uses the following FQDNs to verify the version.

 Expand table

FQDN	Outbound TCP Ports	Usage
<code>aka.ms</code>	443	Vanity URL that provides redirection to the version file
<code>raw.githubusercontent.com</code>	443	The identity service version file hosted in GitHub

Configure communication through a proxy

If your devices are going to be deployed on a network that uses a proxy server, they need to be able to communicate through the proxy to reach IoT Hub and container registries. For more information, see [Configure an IoT Edge device to communicate through a proxy server](#).

Set DNS server in container engine settings

Specify the DNS server for your environment in the container engine settings. The DNS server setting applies to all container modules started by the engine.

1. In the `/etc/docker` directory on your device, edit the `daemon.json` file. Create the file if it doesn't exists.
2. Add the `dns` key and set the DNS server address to a publicly accessible DNS service. If your edge device can't access a public DNS server, use an accessible DNS server address in your network. For example:

JSON

```
{  
    "dns": ["1.1.1.1"]  
}
```

Solution management

- Helpful
 - Set up logs and diagnostics
 - Set up default logging driver
 - Consider tests and CI/CD pipelines

Set up logs and diagnostics

On Linux, the IoT Edge daemon uses journals as the default logging driver. You can use the command-line tool `journalctl` to query the daemon logs.

Starting with version 1.2, IoT Edge relies on multiple daemons. While each daemon's logs can be individually queried with `journalctl`, the `iotedge system` commands provide a convenient way to query the combined logs.

- Consolidated `iotedge` command:

Bash

```
sudo iotedge system logs
```

- Equivalent `journalctl` command:

Bash

```
journalctl -u aziot-edge -u aziot-identityd -u aziot-keyd -u aziot-certd -u aziot-tpmd
```

When you're testing an IoT Edge deployment, you can usually access your devices to retrieve logs and troubleshoot. In a deployment scenario, you may not have that option. Consider how you're going to gather information about your devices in production. One option is to use a logging module that collects information from the other modules and sends it to the cloud. One example of a logging module is [logspout-loganalytics](#), or you can design your own.

Set up default logging driver

By default, the Moby container engine does not set container log size limits. Over time, this can lead to the device filling up with logs and running out of disk space. Configure your container engine to use the [local logging driver](#) as your logging mechanism.

`Local` logging driver offers a default log size limit, performs log-rotation by default, and uses a more efficient file format, which helps to prevent disk space exhaustion. You may also choose to use different [logging drivers](#) and set different size limits based on your need.

Option: Configure the default logging driver for all container modules

You can configure your container engine to use a specific logging driver by setting the value of `log driver` to the name of the log driver in the `daemon.json`. The following example sets the default logging driver to the `local` log driver (recommended).

JSON

```
{  
  "log-driver": "local"  
}
```

You can also configure your `log-opts` keys to use appropriate values in the `daemon.json` file. The following example sets the log driver to `local` and sets the `max-size` and `max-`

file options.

JSON

```
{  
    "log-driver": "local",  
    "log-opt": {  
        "max-size": "10m",  
        "max-file": "3"  
    }  
}
```

Add (or append) this information to a file named `daemon.json` and place it in the following location:

- `/etc/docker/`

The container engine must be restarted for the changes to take effect.

Option: Adjust log settings for each container module

You can do so in the `createOptions` of each module. For example:

yml

```
"createOptions": {  
    "HostConfig": {  
        "LogConfig": {  
            "Type": "local",  
            "Config": {  
                "max-size": "10m",  
                "max-file": "3"  
            }  
        }  
    }  
}
```

Additional options on Linux systems

- Configure the container engine to send logs to `systemd journal` by setting `journald` as the default logging driver.
- Periodically remove old logs from your device by installing a logrotate tool. Use the following file specification:

txt

```
/var/lib/docker/containers/*/*-json.log{
    copytruncate
    daily
    rotate7
    delaycompress
    compress
    notifempty
    missingok
}
```

Consider tests and CI/CD pipelines

For the most efficient IoT Edge deployment scenario, consider integrating your production deployment into your testing and CI/CD pipelines. Azure IoT Edge supports multiple CI/CD platforms, including Azure DevOps. For more information, see [Continuous integration and continuous deployment to Azure IoT Edge](#).

Security considerations

- **Important**
 - Manage access to your container registry
 - Limit container access to host resources

Manage access to your container registry

Before you deploy modules to production IoT Edge devices, ensure that you control access to your container registry so that outsiders can't access or make changes to your container images. Use a private container registry to manage container images.

In the tutorials and other documentation, we instruct you to use the same container registry credentials on your IoT Edge device as you use on your development machine. These instructions are only intended to help you set up testing and development environments more easily, and should not be followed in a production scenario.

For a more secured access to your registry, you have a choice of [authentication options](#). A popular and recommended authentication is to use an Active Directory service principal that's well suited for applications or services to pull container images in an automated or otherwise unattended (headless) manner, as IoT Edge devices do. Another option is to use repository-scoped tokens, which allow you to create long or short-live identities that exist only in the Azure Container Registry they were created in and scope access to the repository level.

To create a service principal, run the two scripts as described in [create a service principal](#). These scripts do the following tasks:

- The first script creates the service principal. It outputs the Service principal ID and the Service principal password. Store these values securely in your records.
- The second script creates role assignments to grant to the service principal, which can be run subsequently if needed. We recommend applying the **acrPull** user role for the `role` parameter. For a list of roles, see [Azure Container Registry roles and permissions](#).

To authenticate using a service principal, provide the service principal ID and password that you obtained from the first script. Specify these credentials in the deployment manifest.

- For the username or client ID, specify the service principal ID.
- For the password or client secret, specify the service principal password.

To create repository-scoped tokens, follow [create a repository-scoped token](#).

To authenticate using repository-scoped tokens, provide the token name and password that you obtained after creating your repository-scoped token. Specify these credentials in the deployment manifest.

- For the username, specify the token's username.
- For the password, specify one of the token's passwords.

Note

After implementing an enhanced security authentication, disable the **Admin user** setting so that the default username/password access is no longer available. In your container registry in the Azure portal, from the left pane menu under **Settings**, select **Access Keys**.

Limit container access to host resources

To balance shared host resources across modules, we recommend putting limits on resource consumption per module. These limits ensure that one module can't consume too much memory or CPU usage and prevent other processes from running on the device. The IoT Edge platform does not limit resources for modules by default, since knowing how much resource a given module needs to run optimally requires testing.

Docker provides some constraints that you can use to limit resources like memory and CPU usage. For more information, see [Runtime options with memory, CPUs, and GPUs](#).

These constraints can be applied to individual modules by using create options in deployment manifests. For more information, see [How to configure container create options for IoT Edge modules](#).

Next steps

- Learn more about [IoT Edge automatic deployment](#).
 - See how IoT Edge supports [Continuous integration and continuous deployment](#).
-

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Develop your own IoT Edge modules

Article • 06/10/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge modules can connect with other Azure services and contribute to your larger cloud data pipeline. This article describes how you can develop modules to communicate with the IoT Edge runtime and IoT Hub, and therefore the rest of the Azure cloud.

IoT Edge runtime environment

The IoT Edge runtime provides the infrastructure to integrate the functionality of multiple IoT Edge modules and to deploy them onto IoT Edge devices. Any program can be packaged as an IoT Edge module. To take full advantage of IoT Edge communication and management functionalities, a program running in a module can use the Azure IoT Device SDK to connect to the local IoT Edge hub.

Packaging your program as an IoT Edge module

To deploy your program on an IoT Edge device, it must first be containerized and run with a Docker-compatible engine. IoT Edge uses [Moby](#), the open-source project behind Docker, as its Docker-compatible engine. The same parameters that you're used to with Docker can be passed to your IoT Edge modules. For more information, see [How to configure container create options for IoT Edge modules](#).

Using the IoT Edge hub

The IoT Edge hub provides two main functionalities: a proxy to IoT Hub and local communications.

Connecting to IoT Edge hub from a module

Connecting to the local IoT Edge hub from a module involves the same connection steps as for any clients. For more information, see [Connecting to the IoT Edge hub](#).

To use IoT Edge routing over AMQP, you can use the `ModuleClient` from the Azure IoT SDK. Create a `ModuleClient` instance to connect your module to the IoT Edge hub running on the device, similar to how `DeviceClient` instances connect IoT devices to IoT Hub. For more information about the `ModuleClient` class and its communication methods, see the API reference for your preferred SDK language: [C#](#), [C](#), [Python](#), [Java](#), or [Node.js](#).

IoT Hub primitives

IoT Hub sees a module instance as similar to a device. A module instance can:

- Send [device-to-cloud messages](#)
- Receive [direct methods](#) targeted specifically at its identity
- Have a module twin that's distinct and isolated from the [device twin](#) and the other module twins of that device

Currently, modules can't receive cloud-to-device messages or use the file upload feature.

When writing a module, you can connect to the IoT Edge hub and use IoT Hub primitives as you would when using IoT Hub with a device application. The only difference between IoT Edge modules and IoT device applications is that with modules you have to refer to the module identity instead of the device identity.

Device-to-cloud messages

An IoT Edge module can send messages to the cloud via the IoT Edge hub that acts as a local broker and propagates messages to the cloud. To enable complex processing of device-to-cloud messages, an IoT Edge module can intercept and process messages sent by other modules or devices to its local IoT Edge hub. The IoT Edge module will then send new messages with processed data. Chains of IoT Edge modules can thus be created to build local processing pipelines.

To send device-to-cloud telemetry messages using routes:

- Use the `Module Client` class of the [Azure IoT SDK](#). Each module has *input* and *output* endpoints.
- Use a send message method from your `Module Client` class to send messages on the output endpoint of your module.

- Set up a route in the edgeHub module of your device to send this output endpoint to IoT Hub.

To process messages using routes:

- Set up a route to send messages coming from another endpoint (module or device) to the input endpoint of your module.
- Listen for messages on the input endpoint of your module. Each time a new message comes back, a callback function is triggered by the Azure IoT SDK.
- Process your message with this callback function and (optionally) send new messages in your module endpoint queue.

 **Note**

To learn more about declaring a route, see [Learn how to deploy modules and establish routes in IoT Edge](#)

Twins

Twins are one of the primitives provided by IoT Hub. There are JSON documents that store state information including metadata, configurations, and conditions. Each module or device has its own twin.

- To get a module twin with the [Azure IoT SDK](#), call the `ModuleClient.getTwin` method.
- To receive a module twin patch with the Azure IoT SDK, implement a callback function and register it with the `ModuleClient.moduleTwinCallback` method from the Azure IoT SDK so that your callback function is triggered each time a twin patch comes in.

Receive direct methods

To receive a direct method with the [Azure IoT SDK](#), implement a callback function and register it with the `ModuleClient.methodCallback` method from the Azure IoT SDK so that your callback function is triggered each time that a direct method comes in.

Language and architecture support

IoT Edge supports multiple operating systems, device architectures, and development languages so you can build the scenario that matches your needs. Use this section to

understand your options for developing custom IoT Edge modules. You can learn more about tooling support and requirements for each language in [Prepare your development and test environment for IoT Edge](#).

Linux

For all languages in the following table, IoT Edge [supports](#) development for AMD64 and most ARM64 Linux containers. There is support for Debian 11 ARM32 containers, as well.

[\[+\] Expand table](#)

Development language	Development tools
C	Visual Studio Code Visual Studio 2019/2022
C#	Visual Studio Code Visual Studio 2019/2022
Java	Visual Studio Code
Node.js	Visual Studio Code
Python	Visual Studio Code

Note

For cross-platform compilation, like compiling an ARM32 IoT Edge module on an AMD64 development machine, you need to configure the development machine to compile code on target device architecture matching the IoT Edge module. For more information about target device architectures, see [Develop Azure IoT Edge modules using Visual Studio Code](#).

Windows

We no longer support Windows containers. [IoT Edge for Linux on Windows](#) is the recommended way to run IoT Edge on Windows devices.

Module security

You should develop your modules with security in mind. To learn more about securing your modules, see [Docker security](#).

To help improve module security, IoT Edge disables some container features by default. You can override the defaults to provide privileged capabilities to your modules if necessary.

Allow elevated Docker permissions

In the config file on an IoT Edge device, there's a parameter called `allow_elevated_docker_permissions`. When set to `true`, this flag allows the `--privileged` flag and any additional capabilities that you define in the `CapAdd` field of the Docker HostConfig in the [container create options](#).

 **Note**

Currently, this flag is true by default, which allows deployments to grant privileged permissions to modules. We recommend that you set this flag to false to improve device security.

Enable CAP_CHOWN and CAP_SETUID

The Docker capabilities `CAP_CHOWN` and `CAP_SETUID` are disabled by default. These capabilities can be used to write to secure files on the host device and potentially gain root access.

If you need these capabilities, you can manually re-enable them using CapADD in the container create options.

Next steps

[Prepare your development and test environment for IoT Edge](#)

[Develop Azure IoT Edge modules using Visual Studio Code](#)

[Debug Azure IoT Edge modules using Visual Studio Code](#)

[Understand and use Azure IoT Hub SDKs](#)

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback ↗

Prepare your development and test environment for IoT Edge

Article • 06/14/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

IoT Edge moves your existing business logic to devices operating at the edge. To prepare your applications and workloads to run as [IoT Edge modules](#), you need to build them as containers. This article provides guidance around how to configure your development environment so that you can successfully create an IoT Edge solution. Once you have your development environment set up, you can learn how to [Develop your own IoT Edge modules](#).

In any IoT Edge solution, there are at least two machines to consider: the IoT Edge device (or devices) that runs the IoT Edge module, and the development machine that builds, tests, and deploys modules. This article focuses primarily on the development machine. For testing purposes, the two machines can be the same. You can run IoT Edge on your development machine and deploy modules to it.

Operating system

IoT Edge runs on a specific set of [supported operating systems](#). When developing for IoT Edge, you can use most operating systems that can run a container engine. The container engine is a requirement on the development machine to build your modules as containers and push them to a container registry.

If your development machine can't run IoT Edge, skip to the [Testing tools](#) section of this article to learn how to test and debug locally.

The operating systems of the development machine and IoT Edge devices don't need to match. However, the container operating system must be consistent with the development machine and the IoT Edge device. For example, you can develop modules

on a Windows machine and deploy them to a Linux device. The Windows machine needs to run Linux containers to build the modules for the Linux device.

Container engine

The central concept of IoT Edge is that you can remotely deploy your business and cloud logic to devices by packaging it into containers. To build containers, you need a container engine on your development machine.

Any container engine compatible with the Open Container Initiative, like Docker, is capable of building IoT Edge module images. Moby is the supported container engine for IoT Edge devices in production. If you are using Ubuntu Core snaps, the Docker snap is serviced by Canonical and supported for production scenarios.

Development tools

The [Azure IoT Edge development tool](#) is a command line tool to develop and test IoT Edge modules. You can create new IoT Edge scenarios, build module images, run modules in a simulator, and monitor messages sent to IoT Hub. The *iotedgedev* tool is the recommended tool for developing IoT Edge modules.

Both Visual Studio and Visual Studio Code have add-on extensions to help develop IoT Edge solutions. These extensions provide language-specific templates to help create and deploy new IoT Edge scenarios. The Azure IoT Edge extensions for Visual Studio and Visual Studio Code help you code, build, deploy, and debug your IoT Edge solutions. You can create an entire IoT Edge solution that contains multiple modules, and the extensions automatically update a deployment manifest template with each new module addition. The extensions also enable management of IoT devices from within Visual Studio or Visual Studio Code. You can deploy modules to a device, monitor the status, and view messages as they arrive at IoT Hub. Finally, both extensions use the IoT EdgeHub dev tool to enable local running and debugging of modules on your development machine.

IoT Edge dev tool

The Azure IoT Edge dev tool simplifies IoT Edge development with command-line abilities. This tool provides CLI commands to develop, debug, and test modules. The IoT Edge dev tool works with your development system, whether you've manually installed the dependencies on your machine or are using the prebuilt [IoT Edge Dev Container](#) to run the *iotedgedev* tool in a container.

For more information and to get started, see [IoT Edge dev tool wiki](#).

Visual Studio Code extension

The Azure IoT Edge extension for Visual Studio Code provides IoT Edge module templates built on programming languages including C, C#, Java, Node.js, and Python. Templates for Azure functions in C# are also included.

Important

The Azure IoT Edge Visual Studio Code extension is in [maintenance mode](#). The `iotedgedev` tool is the recommended tool for developing IoT Edge modules.

For more information and to download, see [Azure IoT Edge for Visual Studio Code](#).

In addition to the IoT Edge extensions, you may find it helpful to install additional extensions for developing. For example, you can use [Docker Support for Visual Studio Code](#) to manage your images, containers, and registries. Additionally, all the major supported languages have extensions for Visual Studio Code that can help when you're developing modules.

The [Azure IoT Hub](#) extension is useful as a companion for the Azure IoT Edge extension.

Visual Studio 2017/2019 extension

The Azure IoT Edge tools for Visual Studio provide an IoT Edge module template built on C# and C.

Important

The Azure IoT Edge Visual Studio extensions are in maintenance mode. The `iotedgedev` tool is the recommended tool for developing IoT Edge modules.

For more information and to download, see [Azure IoT Edge Tools for Visual Studio 2017](#) or [Azure IoT Edge Tools for Visual Studio 2019](#).

Testing tools

Several testing tools exist to help you simulate IoT Edge devices or debug modules more efficiently. The table below shows a high-level comparison between the tools and the following individual sections describe each tool more specifically.

Only the IoT Edge runtime is supported for production deployments, but the following tools allow you to simulate or easily create IoT Edge devices for development and testing purposes. These tools aren't mutually exclusive, but can work together for a complete development experience.

[] Expand table

Tool	Also known as	Supported platforms	Best for
IoT EdgeHub dev tool	iotedgehubdev	Windows, Linux, macOS	Simulating a device to debug modules.
IoT Edge dev container	iotedgecontainer	Windows, Linux, macOS	Developing without installing dependencies.

IoT EdgeHub dev tool

The Azure IoT EdgeHub dev tool provides a local development and debug experience. The tool helps start IoT Edge modules without the IoT Edge runtime so that you can create, develop, test, run, and debug IoT Edge modules and solutions locally. You don't have to push images to a container registry and deploy them to a device for testing.

The IoT EdgeHub dev tool was designed to work in tandem with the Visual Studio and Visual Studio Code extensions, as well as with the IoT Edge dev tool. The dev tool supports inner loop development as well as outer loop testing, so it integrates with other DevOps tools too.

ⓘ Important

The IoT EdgeHub dev tool is in [maintenance mode](#). Consider using a [Linux virtual machine with IoT Edge runtime installed](#), physical device, or [EFLW](#).

For more information and to install, see [Azure IoT EdgeHub dev tool](#).

IoT Edge dev container

The Azure IoT Edge dev container is a Docker container that has all the dependencies that you need for IoT Edge development. This container makes it easy to get started with whichever language you want to develop in, including C#, Python, Node.js, and Java. All you need to install is a container engine, like Docker or Moby, to pull the container to your development machine.

For more information, see [Azure IoT Edge dev container](#).

DevOps tools

When you're ready to develop at-scale solutions for extensive production scenarios, take advantage of modern DevOps principles including automation, monitoring, and streamlined software engineering processes. IoT Edge has extensions to support DevOps tools including Azure DevOps, Azure DevOps Projects, and Jenkins. If you want to customize an existing pipeline or use a different DevOps tool like CircleCI or TravisCI, you can do so with the CLI features included in the IoT Edge dev tool.

For more information, guidance, and examples, see the following pages:

- [Continuous integration and continuous deployment to Azure IoT Edge](#)
- [IoT Edge DevOps GitHub repo](#)

Feedback

Was this page helpful?



[Provide product feedback](#)

Learn how to deploy modules and establish routes in IoT Edge

Article • 06/05/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Each IoT Edge device runs at least two modules: \$edgeAgent and \$edgeHub, which are part of the IoT Edge runtime. IoT Edge device can run multiple modules for any number of processes. Use a deployment manifest to tell your device which modules to install and how to configure them to work together.

The *deployment manifest* is a JSON document that describes:

- The **IoT Edge agent** module twin, which includes three components:
 - The container image for each module that runs on the device.
 - The credentials to access private container registries that contain module images.
 - Instructions for how each module should be created and managed.
- The **IoT Edge hub** module twin, which includes how messages flow between modules and eventually to IoT Hub.
- The desired properties of any extra module twins (optional).

All IoT Edge devices must be configured with a deployment manifest. A newly installed IoT Edge runtime reports an error code until configured with a valid manifest.

In the Azure IoT Edge tutorials, you build a deployment manifest by going through a wizard in the Azure IoT Edge portal. You can also apply a deployment manifest programmatically using REST or the IoT Hub Service SDK. For more information, see [Understand IoT Edge deployments](#).

Create a deployment manifest

At a high level, a deployment manifest is a list of module twins that are configured with their desired properties. A deployment manifest tells an IoT Edge device (or a group of

devices) which modules to install and how to configure them. Deployment manifests include the *desired properties* for each module twin. IoT Edge devices report back the *reported properties* for each module.

Two modules are required in every deployment manifest: `$edgeAgent`, and `$edgeHub`. These modules are part of the IoT Edge runtime that manages the IoT Edge device and the modules running on it. For more information about these modules, see [Understand the IoT Edge runtime and its architecture](#).

In addition to the two runtime modules, you can add up to 50 modules of your own to run on an IoT Edge device.

A deployment manifest that contains only the IoT Edge runtime (`edgeAgent` and `edgeHub`) is valid.

Deployment manifests follow this structure:

```
JSON

{
    "modulesContent": {
        "$edgeAgent": { // required
            "properties.desired": {
                // desired properties of the IoT Edge agent
                // includes the image URIs of all deployed modules
                // includes container registry credentials
            }
        },
        "$edgeHub": { //required
            "properties.desired": {
                // desired properties of the IoT Edge hub
                // includes the routing information between modules, and to IoT Hub
            }
        },
        "module1": { // optional
            "properties.desired": {
                // desired properties of module1
            }
        },
        "module2": { // optional
            "properties.desired": {
                // desired properties of module2
            }
        }
    }
}
```

Configure modules

Define how the IoT Edge runtime installs the modules in your deployment. The IoT Edge agent is the runtime component that manages installation, updates, and status reporting for an IoT Edge device. Therefore, the \$edgeAgent module twin contains the configuration and management information for all modules. This information includes the configuration parameters for the IoT Edge agent itself.

The \$edgeAgent properties follow this structure:

```
JSON

{
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.1",
        "runtime": {
          "settings": {
            "registryCredentials": {
              // give the IoT Edge agent access to container images that
aren't public
            }
          }
        },
        "systemModules": {
          "edgeAgent": {
            // configuration and management details
          },
          "edgeHub": {
            // configuration and management details
          }
        },
        "modules": {
          "module1": {
            // configuration and management details
          },
          "module2": {
            // configuration and management details
          }
        }
      }
    },
    "$edgeHub": { ... },
    "module1": { ... },
    "module2": { ... }
  }
}
```

The IoT Edge agent schema version 1.1 was released along with IoT Edge version 1.0.10, and enables module startup order. Schema version 1.1 is recommended for any IoT Edge deployment running version 1.0.10 or later.

Module configuration and management

The IoT Edge agent desired properties list is where you define which modules are deployed to an IoT Edge device and how they should be configured and managed.

For a complete list of desired properties that can or must be included, see [Properties of the IoT Edge agent and IoT Edge hub](#).

For example:

```
JSON

{
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.1",
        "runtime": { ... },
        "systemModules": {
          "edgeAgent": { ... },
          "edgeHub": { ... }
        },
        "modules": {
          "module1": {
            "version": "1.0",
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "startupOrder": 2,
            "settings": {
              "image": "myacr.azurecr.io/module1:latest",
              "createOptions": "{}"
            }
          },
          "module2": { ... }
        }
      }
    },
    "$edgeHub": { ... },
    "module1": { ... },
    "module2": { ... }
  }
}
```

Every module has a **settings** property that contains the module **image**, an address for the container image in a container registry, and any **createOptions** to configure the image on startup. For more information, see [How to configure container create options for IoT Edge modules](#).

The edgeHub module and custom modules also have three properties that tell the IoT Edge agent how to manage them:

- **Status:** Whether the module should be running or stopped when first deployed. Required.
- **RestartPolicy:** When and if the IoT Edge agent should restart the module if it stops. If the module is stopped without any errors, it won't start automatically. For more information, see [Docker Docs - Start containers automatically](#). Required.
- **StartupOrder:** *Introduced in IoT Edge version 1.0.10.* Which order the IoT Edge agent should start the modules when first deployed. The order is declared with integers, where a module given a startup value of 0 is started first and then higher numbers follow. The edgeAgent module doesn't have a startup value because it always starts first. Optional.

The IoT Edge agent initiates the modules in order of the startup value, but doesn't wait for each module to finish starting before going to the next one.

Startup order is helpful if some modules depend on others. For example, you may want the edgeHub module to start first so that it's ready to route messages when the other modules start. Or you may want to start a storage module before you start modules that send data to it. However, you should always design your modules to handle failures of other modules. It's the nature of containers that they may stop and restart at any time, and any number of times.

Note

Changes to a module's properties will result in that module restarting. For example, a restart will happen if you change properties for the:

- module image
- Docker create options
- environment variables
- restart policy
- image pull policy
- version
- startup order

If no module property is changed, the module will **not** restart.

Declare routes

The IoT Edge hub manages communication between modules, IoT Hub, and any downstream devices. Therefore, the `$edgeHub` module twin contains a desired property called *routes* that declares how messages are passed within a deployment. You can have multiple routes within the same deployment.

Routes are declared in the `$edgeHub` desired properties with the following syntax:

```
JSON

{
    "modulesContent": {
        "$edgeAgent": { ... },
        "$edgeHub": {
            "properties.desired": {
                "schemaVersion": "1.1",
                "routes": {
                    "route1": "FROM <source> WHERE <condition> INTO <sink>",
                    "route2": {
                        "route": "FROM <source> WHERE <condition> INTO <sink>",
                        "priority": 0,
                        "timeToLiveSecs": 86400
                    }
                },
                "storeAndForwardConfiguration": {
                    "timeToLiveSecs": 10
                }
            }
        },
        "module1": { ... },
        "module2": { ... }
    }
}
```

The IoT Edge hub schema version 1 was released along with IoT Edge version 1.0.10, and enables route prioritization and time to live. Schema version 1.1 is recommended for any IoT Edge deployment running version 1.0.10 or later.

Every route needs a *source* where the messages come from and a *sink* where the messages go. The *condition* is an optional piece that you can use to filter messages.

You can assign *priority* to routes that you want to make sure process their messages first. This feature is helpful in scenarios where the upstream connection is weak or limited and you have critical data that should be prioritized over standard telemetry messages.

Source

The source specifies where the messages come from. IoT Edge can route messages from modules or downstream devices.

With the IoT SDKs, modules can declare specific output queues for their messages using the `ModuleClient` class. Output queues aren't necessary, but are helpful for managing multiple routes. Downstream devices can use the `DeviceClient` class of the IoT SDKs to send messages to IoT Edge gateway devices in the same way that they would send messages to IoT Hub. For more information, see [Understand and use Azure IoT Hub SDKs](#).

The source property can be any of the following values:

[+] [Expand table](#)

Source	Description
<code>/*</code>	All device-to-cloud messages or twin change notifications from any module or downstream device
<code>/twinChangeNotifications</code>	Any twin change (reported properties) coming from any module or downstream device
<code>/messages/*</code>	Any device-to-cloud message sent by a module through some or no output, or by a downstream device
<code>/messages/modules/*</code>	Any device-to-cloud message sent by a module through some or no output
<code>/messages/modules/<moduleId>/*</code>	Any device-to-cloud message sent by a specific module through some or no output
<code>/messages/modules/<moduleId>/outputs/*</code>	Any device-to-cloud message sent by a specific module through some output
<code>/messages/modules/<moduleId>/outputs/<output></code>	Any device-to-cloud message sent by a specific module through a specific output

Condition

The condition is optional in a route declaration. If you want to pass all messages from the source to the sink, just leave out the `WHERE` clause entirely. Or you can use the [IoT Hub query language](#) to filter for certain messages or message types that satisfy the condition. IoT Edge routes don't support filtering messages based on twin tags or properties.

The messages that pass between modules in IoT Edge are formatted the same as the messages that pass between your devices and Azure IoT Hub. All messages are formatted as JSON and have **systemProperties**, **appProperties**, and **body** parameters.

You can build queries around any of the three parameters with the following syntax:

- System properties: `$<propertyName>` or `{$<propertyName>}`
- Application properties: `<propertyName>`
- Body properties: `$body.<propertyName>`

For examples about how to create queries for message properties, see [Device-to-cloud message routes query expressions](#).

An example that is specific to IoT Edge is when you want to filter for messages that arrived at a gateway device from a downstream device. Messages sent from modules include a system property called **connectionModuleId**. So if you want to route messages from downstream devices directly to IoT Hub, use the following route to exclude module messages:

```
query
```

```
FROM /messages/* WHERE NOT IS_DEFINED($connectionModuleId) INTO $upstream
```

Sink

The sink defines where the messages are sent. Only modules and IoT Hub can receive messages. Messages can't be routed to other devices. There are no wildcard options in the sink property.

The sink property can be any of the following values:

[\[+\] Expand table](#)

Sink	Description
<code>\$upstream</code>	Send the message to IoT Hub
<code>BrokeredEndpoint("/modules/<moduleId>/inputs/<input>")</code>	Send the message to a specific input of a specific module

IoT Edge provides at-least-once guarantees. The IoT Edge hub stores messages locally in case a route can't deliver the message to its sink. For example, if the IoT Edge hub can't connect to IoT Hub, or the target module isn't connected.

IoT Edge hub stores the messages up to the time specified in the `storeAndForwardConfiguration.timeToLiveSecs` property of the IoT Edge hub desired properties.

Priority and time-to-live

Routes can be declared with either just a string defining the route, or as an object that takes a route string, a priority integer, and a time-to-live integer.

Option 1:

JSON

```
"route1": "FROM <source> WHERE <condition> INTO <sink>",
```

Option 2, introduced in IoT Edge version 1.0.10 with IoT Edge hub schema version 1.1:

JSON

```
"route2": {  
    "route": "FROM <source> WHERE <condition> INTO <sink>",  
    "priority": 0,  
    "timeToLiveSecs": 86400  
}
```

Priority values can be 0-9, inclusive, where 0 is the highest priority. Messages are queued up based on their endpoints. All priority 0 messages targeting a specific endpoint are processed before any priority 1 messages targeting the same endpoint are processed, and down the line. If multiple routes for the same endpoint have the same priority, their messages are processed in a first-come-first-served basis. If no priority is specified, the route is assigned to the lowest priority.

The `timeToLiveSecs` property inherits its value from IoT Edge hub's `storeAndForwardConfiguration` unless explicitly set. The value can be any positive integer.

For detailed information about how priority queues are managed, see the reference page for [Route priority and time-to-live](#).

Define or update desired properties

The deployment manifest specifies desired properties for each module deployed to the IoT Edge device. Desired properties in the deployment manifest overwrite any desired

properties currently in the module twin.

If you don't specify a module twin's desired properties in the deployment manifest, IoT Hub won't modify the module twin in any way. Instead, you can set the desired properties programmatically.

The same mechanisms that allow you to modify device twins are used to modify module twins. For more information, see the [module twin developer guide](#).

Deployment manifest example

The following example shows what a valid deployment manifest document may look like.

```
JSON

{
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.1",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": "",
            "registryCredentials": {
              "ContosoRegistry": {
                "username": "myacr",
                "password": "<password>",
                "address": "myacr.azurecr.io"
              }
            }
          }
        },
        "systemModules": {
          "edgeAgent": {
            "type": "docker",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-agent:1.5",
              "createOptions": "{}"
            }
          },
          "edgeHub": {
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "startupOrder": 0,
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-hub:1.5",
              "portMappings": [
                {
                  "hostPort": 8883,
                  "containerPort": 8883,
                  "protocol": "tcp"
                }
              ]
            }
          }
        }
      }
    }
  }
}
```

```

        "createOptions": "{\"HostConfig\":{\"PortBindings\":
        {\"443/tcp\":[{\"HostPort\":\"443\"}],\"5671/tcp\":
        [{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"]}]}"
    }
},
"modules": {
    "SimulatedTemperatureSensor": {
        "version": "1.5",
        "type": "docker",
        "status": "running",
        "restartPolicy": "always",
        "startupOrder": 2,
        "settings": {
            "image": "mcr.microsoft.com/azureiotedge-simulated-
temperature-sensor:1.5",
            "createOptions": "{}"
        }
    },
    "filtermodule": {
        "version": "1.0",
        "type": "docker",
        "status": "running",
        "restartPolicy": "always",
        "startupOrder": 1,
        "env": {
            "tempLimit": {"value": "100"}
        },
        "settings": {
            "image": "myacr.azurecr.io/filtermodule:latest",
            "createOptions": "{}"
        }
    }
}
},
"$edgeHub": {
    "properties.desired": {
        "schemaVersion": "1.1",
        "routes": {
            "sensorToFilter": {
                "route": "FROM
/messages/modules/SimulatedTemperatureSensor/outputs/temperatureOutput INTO
BrokeredEndpoint(\"/modules/filtermodule/inputs/input1\"),

                "priority": 0,
                "timeToLiveSecs": 1800
            },
            "filterToIoTHub": {
                "route": "FROM /messages/modules/filtermodule/outputs/output1
INTO $upstream",
                "priority": 1,
                "timeToLiveSecs": 1800
            }
        },
        "storeAndForwardConfiguration": {

```

```
        "timeToLiveSecs": 100
    }
}
}
}
```

Next steps

- For a complete list of properties that can or must be included in \$edgeAgent and \$edgeHub, see [Properties of the IoT Edge agent and IoT Edge hub](#).
- Now that you know how IoT Edge modules are used, [Understand the requirements and tools for developing IoT Edge modules](#).

Understand IoT Edge automatic deployments for single devices or at scale

Article • 06/10/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Automatic deployments and layered deployment help you manage and configure modules on large numbers of IoT Edge devices.

Azure IoT Edge provides two ways to configure the modules to run on IoT Edge devices. The first method is to deploy modules on a per-device basis. You create a deployment manifest and then apply it to a particular device by name. The second method is to deploy modules automatically to any registered device that meets a set of defined conditions. You create a deployment manifest and then define which devices it applies to based on [tags](#) in the device twin.

You can't combine per-device and automatic deployments. Once you start targeting IoT Edge devices with automatic deployments (with or without layered deployments), per-device deployments are no longer supported.

This article focuses on configuring and monitoring fleets of devices, collectively referred to as *IoT Edge automatic deployments*.

The basic deployment steps are as follows:

1. An operator defines a deployment manifest that describes a set of modules and the target devices.
2. As a result, the IoT Hub service communicates with all targeted devices to configure them with the declared modules.
3. The IoT Hub service retrieves status from the IoT Edge devices and makes them available to the operator. For example, an operator can see when an Edge device isn't configured successfully or if a module fails during runtime.

4. At any time, when newly targeted IoT Edge devices come online and connect with IoT Hub, they're configured for the deployment.

This article describes each component involved in configuring and monitoring a deployment. For a walkthrough of creating and updating a deployment, see [Deploy and monitor IoT Edge modules at scale](#).

Deployment

An IoT Edge automatic deployment assigns IoT Edge module images to run as instances on a targeted set of IoT Edge devices. The automated deployment configures an IoT Edge deployment manifest to include a list of modules with the corresponding initialization parameters. A deployment can be assigned to a single device (based on Device ID) or to a group of devices (based on tags). Once an IoT Edge device receives a deployment manifest, it downloads and installs the container images from the respective container repositories, and configures them accordingly. Once a deployment is created, an operator can monitor the deployment status to see whether targeted devices are correctly configured.

Only IoT Edge devices can be configured with a deployment. The following prerequisites must be on the device before it can receive the deployment:

- The base operating system
- A container management system, like Moby or Docker
- Provisioning of the IoT Edge runtime

Deployment manifest

A deployment manifest is a JSON document that describes the modules to be configured on the targeted IoT Edge devices. It contains the configuration metadata for all the modules, including the required system modules (specifically the IoT Edge agent and IoT Edge hub).

The configuration metadata for each module includes:

- Version
- Type
- Status (for example, *Running* or *Stopped*)
- Restart policy
- Image and container registry
- Routes for data input and output

If the module image is stored in a private container registry, the IoT Edge agent holds the registry credentials.

Target condition

The target device condition is continuously evaluated throughout the lifetime of the deployment. Any new devices that meet the requirements are included, and any existing devices that no longer meet requirements are removed. The deployment is reactivated if the service detects any target condition change.

For example, you have a deployment with a target condition `tags.environment = 'prod'`. When you initiate the deployment, there are 10 production devices. The modules are successfully installed in these 10 devices. The IoT Edge agent status shows 10 total devices, 10 successful responses, 0 failure responses, and 0 pending responses. Now you add five more devices with `tags.environment = 'prod'`. The service detects the change and the IoT Edge agent status now shows 15 total devices, 10 successful responses, 0 failure responses, and 5 pending responses while it deploys to the five new devices.

If a deployment has no target condition, then it's applied to no devices.

Use any Boolean condition on device twin tags, device twin reported properties, or deviceId to select the target devices. If you want to use a condition with tags, you need to add a `"tags":{}` section in the device twin under the same level as properties. For more information about tags in a device twin, see [Understand and use device twins in IoT Hub](#). For more information about query operations, see [IoT Hub query language operators and IS_DEFINED function](#).

Examples of target conditions:

- `deviceId ='linuxprod1'`
- `tags.environment ='prod'`
- `tags.environment = 'prod' AND tags.location = 'westus'`
- `tags.environment = 'prod' OR tags.location = 'westus'`
- `tags.operator = 'John' AND tags.environment = 'prod' AND NOT deviceId = 'linuxprod1'`
- `properties.reported.devicemodel = '4000x'`
- `IS_DEFINED(tags.remote)`
- `NOT IS_DEFINED(tags.location.building)`
- `tags.environment != null`
- `[none]`

Consider these constraints when you construct a target condition:

- In the device twin, you can only build a target condition using tags, reported properties, or deviceld.
- Double quotes aren't allowed in any portion of the target condition. Use single quotes.
- Single quotes represent the values of the target condition. Therefore, you must escape the single quote with another single quote if it's part of the device name. For example, to target a device called `operator'sDevice`, write
`deviceId='operator''sDevice'.`
- Numbers, letters, and the following characters are allowed in target condition values: `"()<>@,;:\\"/?={}\t\n\r.`
- The following characters aren't allowed in target condition keys: `/;.`

Priority

A priority defines whether a deployment should be applied to a targeted device relative to other deployments. A deployment priority is a positive integer within the range from 0 through 2,147,483,647. Larger numbers denote a higher priority. If an IoT Edge device is targeted by more than one deployment, the deployment with the highest priority applies. Deployments with lower priorities aren't applied, nor are they merged. If a device is targeted with two or more deployments with equal priority, the most recently created deployment (determined by the creation timestamp) applies.

Labels

Labels are string key/value pairs that you can use to filter and group deployments. A deployment may have multiple labels. Labels are optional and don't impact the configuration of IoT Edge devices.

Metrics

By default, all deployments report on four metrics:

- **Targeted** shows the IoT Edge devices that match the Deployment targeting condition.
- **Applied** shows the targeted IoT Edge devices that aren't targeted by another deployment of higher priority.
- **Reporting Success** shows the IoT Edge devices that report their modules as deployed successfully.

- **Reporting Failure** shows the IoT Edge devices that report one or more modules as deployed unsuccessfully. To further investigate the error, connect remotely to those devices and view the log files.

Additionally, you can define your own custom metrics to help monitor and manage the deployment.

Metrics provide summary counts of the various states that devices may report back as a result of applying a deployment configuration. Metrics can query [edgeHub module twin reported properties](#), like *lastDesiredStatus* or *lastConnectTime*.

For example:

SQL

```
SELECT deviceId FROM devices
WHERE properties.reported.lastDesiredStatus.code = 200
```

Adding your own metrics is optional, and doesn't impact the actual configuration of IoT Edge devices.

Layered deployment

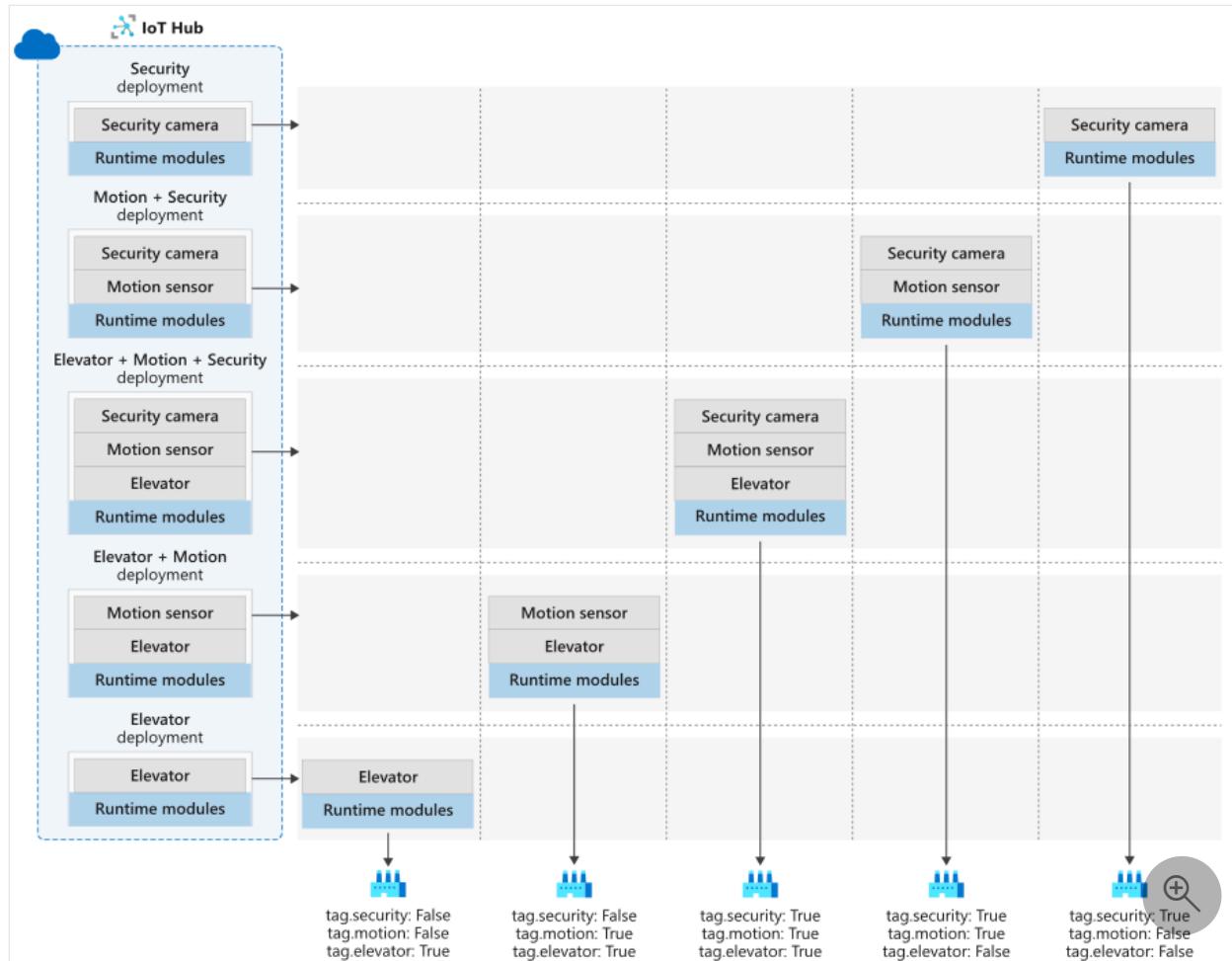
Layered deployments are automatic deployments that can be combined together to reduce the number of unique deployments that need to be created. Layered deployments are useful in scenarios where the same modules are reused in different combinations in many automatic deployments.

Layered deployments have the same basic components as any automatic deployment. They target devices based on tags in the device twins and provide the same functionality around labels, metrics, and status reporting. Layered deployments also have priorities assigned to them. Instead of using the priority to determine which deployment is applied to a device, the priority determines how multiple deployments are ranked on a device. For example, if two layered deployments have a module or a route with the same name, the layered deployment with the higher priority will be applied while the lower priority is overwritten.

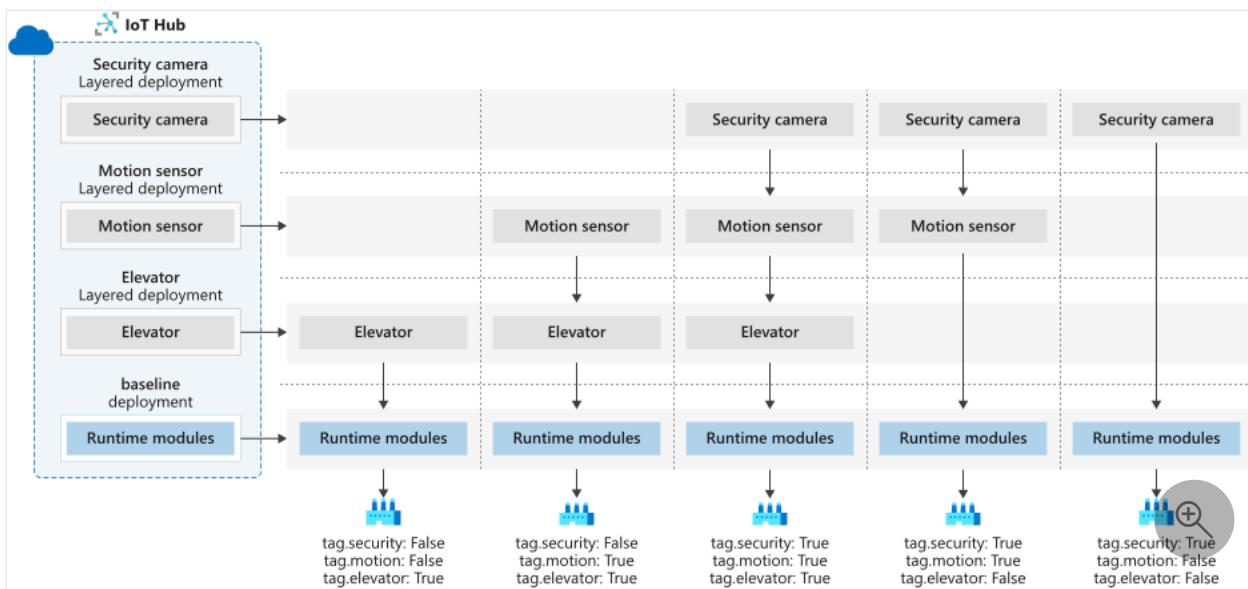
The system runtime modules, known as `edgeAgent` and `edgeHub`, are not configured as part of a layered deployment. Any IoT Edge device targeted by a layered deployment, first needs a standard automatic deployment applied to it. The automatic deployment provides the base upon which layered deployments can be added.

An IoT Edge device can apply one and only one standard automatic deployment, but it can apply multiple layered automatic deployments. Any layered deployments targeting a device must have a higher priority than the automatic deployment for that device.

For example, consider the following scenario of a company that manages buildings. The company developed IoT Edge modules for collecting data from security cameras, motion sensors, and elevators. However, not all their buildings can use all three modules. With standard automatic deployments, the company needs to create individual deployments for all the module combinations that their buildings need.



However, once the company switches to layered automatic deployments, they can create the same module combinations for their buildings with fewer deployments to manage. Each module has its own layered deployment, and the device tags identify which modules get added to each building.



Module twin configuration

When you work with layered deployments, you may, intentionally or otherwise, have two deployments with the same module targeting a device. In those cases, you can decide whether the higher priority deployment should overwrite the module twin or append to it. For example, you may have a deployment that applies the same module to 100 different devices. However, 10 of those devices are in secure facilities and need additional configuration in order to communicate through proxy servers. You can use a layered deployment to add module twin properties that enable those 10 devices to communicate securely without overwriting the existing module twin information from the base deployment.

You can append module twin desired properties in the deployment manifest. In a standard deployment, you would add properties in the `properties.desired` section of the module twin. But in a layered deployment, you can declare a new subset of desired properties.

For example, in a standard deployment you might add the simulated temperature sensor module with the following desired properties that tell it to send data in 5-second intervals:

```
JSON
{
  "SimulatedTemperatureSensor": {
    "properties.desired": {
      "SendData": true,
      "SendInterval": 5
    }
  }
}
```

In a layered deployment that targets some or all of these same devices, you could add a property that tells the simulated sensor to send 1000 messages and then stop. You don't want to overwrite the existing properties, so you create a new section within the desired properties called `layeredProperties`, which contains the new property:

JSON

```
"SimulatedTemperatureSensor": {  
    "properties.desired.layeredProperties": {  
        "StopAfterCount": 1000  
    }  
}
```

A device that has both deployments applied will reflect the following properties in the module twin for the simulated temperature sensor:

JSON

```
"properties": {  
    "desired": {  
        "SendData": true,  
        "SendInterval": 5,  
        "layeredProperties": {  
            "StopAfterCount": 1000  
        }  
    }  
}
```

If you set the `properties.desired` field of the module twin in a layered deployment, `properties.desired` will overwrite the desired properties for that module in any lower priority deployments.

Phased rollout

A phased rollout is an overall process whereby an operator deploys changes to a broadening set of IoT Edge devices. The goal is to make changes gradually to reduce the risk of making wide-scale breaking changes. Automatic deployments help manage phased rollouts across a fleet of IoT Edge devices.

A phased rollout is executed in the following phases and steps:

1. Establish a test environment of IoT Edge devices by provisioning them and setting a device twin tag like `tag.environment='test'`. The test environment should mirror the production environment that the deployment will eventually target.

2. Create a deployment including the desired modules and configurations. The targeting condition should target the test IoT Edge device environment.
3. Validate the new module configuration in the test environment.
4. Update the deployment to include a subset of production IoT Edge devices by adding a new tag to the targeting condition. Also, ensure that the priority for the deployment is higher than other deployments currently targeted to those devices.
5. Verify that the deployment succeeded on the targeted IoT Edge devices by viewing the deployment status.
6. Update the deployment to target all remaining production IoT Edge devices.

Rollback

Deployments can be rolled back if you receive errors or misconfigurations. Because a deployment defines the absolute module configuration for an IoT Edge device, an additional deployment must also be targeted to the same device at a lower priority even if the goal is to remove all modules.

Deleting a deployment doesn't remove the modules from targeted devices. There must be another deployment that defines a new configuration for the devices, even if it's an empty deployment.

However, deleting a deployment may remove modules from the targeted device if it was a layered deployment. A layered deployment updates the underlying deployment, potentially adding modules. Removing a layered deployment removes its update to the underlying deployment, potentially removing modules.

For example, a device has base deployment A and layered deployments O and M applied onto it (so that the A, O, and M deployments are deployed onto the device). If layered deployment M is then deleted, A and O are applied onto the device, and the modules unique to deployment M are removed.

Perform rollbacks in the following sequence:

1. Confirm that a second deployment is also targeted at the same device set. If the goal of the rollback is to remove all modules, the second deployment should not include any modules.
2. Modify or remove the target condition expression of the deployment you wish to roll back so that the devices no longer meet the targeting condition.
3. Verify that the rollback succeeded by viewing the deployment status.
 - The rolled-back deployment should no longer show status for the devices that were rolled back.

- The second deployment should now include deployment status for the devices that were rolled back.

Next steps

- Walk through the steps to create, update, or delete a deployment in [Deploy and monitor IoT Edge modules at scale](#).
 - Learn more about other IoT Edge concepts like the [IoT Edge runtime](#) and [IoT Edge modules](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Security standards for Azure IoT Edge

Article • 07/28/2023

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge addresses the risks that are inherent when moving your data and analytics to the intelligent edge. The IoT Edge security standards balance flexibility for different deployment scenarios with the protection that you expect from all Azure services.

IoT Edge runs on various makes and models of hardware, supports several operating systems, and applies to diverse deployment scenarios. Rather than offering concrete solutions for specific scenarios, IoT Edge is an extensible security framework based on well-grounded principles that are designed for scale. The risk of a deployment scenario depends on many factors, including:

- Solution ownership
- Deployment geography
- Data sensitivity
- Privacy
- Application vertical
- Regulatory requirements

This article provides an overview of the IoT Edge security framework. For more information, see [Securing the intelligent edge](#).

Standards

Standards promote ease of scrutiny and ease of implementation, both of which are hallmarks of security. A security solution should lend itself to scrutiny under evaluation to build trust and shouldn't be a hurdle to deployment. The design of the framework to secure Azure IoT Edge is based on time-tested and industry proven security protocols for familiarity and reuse.

Authentication

When you deploy an IoT solution, you need to know that only trusted actors, devices, and modules have access to your solution. Certificate-based authentication is the primary mechanism for authentication for the Azure IoT Edge platform. This mechanism is derived from a set of standards governing Public Key Infrastructure (PKIX) by the Internet Engineering Task Force (IETF).

All devices, modules, and actors that interact with the Azure IoT Edge device should have unique certificate identities. This guidance applies whether the interactions are physical or through a network connection. Not every scenario or component may lend itself to certificate-based authentication, so the extensibility of the security framework offers secure alternatives.

For more information, see [Azure IoT Edge certificate usage](#).

Authorization

The principle of least privilege says that users and components of a system should have access only to the minimum set of resources and data needed to perform their roles. Devices, modules, and actors should access only the resources and data within their permission scope, and only when it's architecturally allowable. Some permissions are configurable with sufficient privileges and others are architecturally enforced. For example, some modules may be authorized to connect to Azure IoT Hub. However, there's no reason why a module in one IoT Edge device should access the twin of a module in another IoT Edge device.

Other authorization schemes include certificate signing rights and role-based access control (RBAC).

Attestation

Attestation ensures the integrity of software bits, which is important for detecting and preventing malware. The Azure IoT Edge security framework classifies attestation under three main categories:

- Static attestation
- Runtime attestation
- Software attestation

Static attestation

Static attestation verifies the integrity of all software on a device during power-up, including the operating system, all runtimes, and configuration information. Because static attestation occurs during power-up, it's often referred to as secure boot. The security framework for IoT Edge devices extends to manufacturers and incorporates secure hardware capabilities that assure static attestation processes. These processes include secure boot and secure firmware upgrade. Working in close collaboration with silicon vendors eliminates superfluous firmware layers, so minimizes the threat surface.

Runtime attestation

Once a system has completed a secure boot process, well-designed systems should detect attempts to inject malware and take proper countermeasures. Malware attacks may target the system's ports and interfaces. If malicious actors have physical access to a device, they may tamper with the device itself or use side-channel attacks to gain access. Such malcontent, whether malware or unauthorized configuration changes, can't be detected by static attestation because it's injected after the boot process. Countermeasures offered or enforced by the device's hardware help to ward off such threats. The security framework for IoT Edge explicitly calls for extensions that combat runtime threats.

Software attestation

All healthy systems, including intelligent edge systems, need patches and upgrades. Security is important for update processes, otherwise they can be potential threat vectors. The security framework for IoT Edge calls for updates through measured and signed packages to assure the integrity of and authenticate the source of the packages. This standard applies to all operating systems and application software bits.

Hardware root of trust

For many intelligent edge devices, especially devices that can be physically accessed by potential malicious actors, hardware security is the last defense for protection. Tamper resistant hardware is crucial for such deployments. Azure IoT Edge encourages secure silicon hardware vendors to offer different flavors of hardware root of trust to accommodate various risk profiles and deployment scenarios. Hardware trust may come from common security protocol standards like Trusted Platform Module (ISO/IEC 11889) and Trusted Computing Group's Device Identifier Composition Engine (DICE). Secure enclave technologies like TrustZones and Software Guard Extensions (SGX) also provide hardware trust.

Certification

To help customers make informed decisions when procuring Azure IoT Edge devices for their deployment, the IoT Edge framework includes certification requirements.

Foundational to these requirements are certifications pertaining to security claims and certifications pertaining to validation of the security implementation. For example, a security claim certification means that the IoT Edge device uses secure hardware known to resist boot attacks. A validation certification means that the secure hardware was properly implemented to offer this value in the device. In keeping with the principle of simplicity, the framework tries to keep the burden of certification minimal.

Encryption at rest

Encryption at rest provides data protection for stored data. Attacks against data at-rest include attempts to get physical access to the hardware where the data is stored, and then compromise the contained data. You can use storage encryption to protect data stored on the device. Linux has several options for encryption at rest. Choose the option that best fits your needs. For Windows, [Windows BitLocker](#) is the recommended option for encryption at rest.

Extensibility

With IoT technology driving different types of business transformations, security should evolve in parallel to address emerging scenarios. The Azure IoT Edge security framework starts with a solid foundation on which it builds in extensibility into different dimensions to include:

- First party security services like the Device Provisioning Service for Azure IoT Hub.
- Third-party services like managed security services for different application verticals (like industrial or healthcare) or technology focus (like security monitoring in mesh networks, or silicon hardware attestation services) through a rich network of partners.
- Legacy systems to include retrofitting with alternate security strategies, like using secure technology other than certificates for authentication and identity management.
- Secure hardware for adoption of emerging secure hardware technologies and silicon partner contributions.

In the end, securing the intelligent edge requires collaborative contributions from an open community driven by the common interest in securing IoT. These contributions might be in the form of secure technologies or services. The Azure IoT Edge security

framework offers a solid foundation for security that is extensible for the maximum coverage to offer the same level of trust and integrity in the intelligent edge as with Azure cloud.

Next steps

Read more about how Azure IoT Edge is [Securing the intelligent edge](#).

Azure IoT Edge security manager

Article • 06/06/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The Azure IoT Edge security manager is a well-bounded security core for protecting the IoT Edge device and all its components by abstracting the secure silicon hardware. The security manager is the focal point for security hardening and provides technology integration point to original equipment manufacturers (OEM).

The security manager abstracts the secure silicon hardware on an IoT Edge device and provides an extensibility framework for additional security services.

The IoT Edge security manager aims to defend the integrity of the IoT Edge device and all inherent software operations. The security manager transitions trust from underlying hardware root of trust hardware (if available) to bootstrap the IoT Edge runtime and monitor ongoing operations. The IoT Edge security manager is software working along with secure silicon hardware (where available) to help deliver the highest security assurances possible.

Additionally, the IoT Edge security manager provides a safe framework for security service extensions through host-level modules. These services include security monitoring and updates that require agents inside the device with privileged access to some components of the device. The extensibility framework ensures that such integrations consistently uphold overall system security.

The responsibilities of the IoT Edge security manager include, but aren't limited to:

- Bootstrap the Azure IoT Edge device.
- Control access to the device hardware root of trust through notary services.
- Monitor the integrity of IoT Edge operations at runtime.
- Provision the device identity and manage transition of trust where applicable.
- Ensure safe operation of client agents for services including Device Update for IoT Hub and Microsoft Defender for IoT.

The IoT Edge security manager consists of three components:

- The IoT Edge module runtime
- Hardware security module (HSM) abstractions through standard implementations such as PKCS#11 and Trusted Platform Module (TPM)
- A hardware silicon root of trust or HSM (optional, but highly recommended)

Changes in version 1.2 and later

In versions 1.0 and 1.1 of IoT Edge, a component called the **security daemon** was responsible for the logical security operations of the security manager. In the update to version 1.2, several key responsibilities were delegated to the [Azure IoT Identity Service](#) security subsystem. Once these security-based tasks were removed from the security daemon, its name no longer made sense. To better reflect the work that this component does in version 1.2 and beyond, we renamed it to the **module runtime**.

The IoT Edge module runtime

The IoT Edge module runtime delegates trust from the [Azure IoT Identity Service](#) security subsystem to protect the IoT Edge container runtime environment. One service, now delegated to Azure IoT Identity Service, is the automated certificate enrollment and renewal service through an EST server. To see how this works and create a sample EST server made for an IoT Edge device, try the [Configure Enrollment over Secure Transport Server for Azure IoT Edge](#) tutorial.

The module runtime is responsible for the logical security operations of the security manager. It represents a significant portion of the trusted computing base of the IoT Edge device. The module runtime uses security services from the IoT Identity Service, which is in turn hardened by the device manufacturer's choice of hardware security module (HSM). We strongly recommend the use of HSMs for device hardening.

Design principles

IoT Edge follows two core principles: maximize operational integrity, and minimize bloat and churn.

Maximize operational integrity

The IoT Edge module runtime operates with the highest integrity possible within the defense capability of any given root of trust hardware. With proper integration, the root of trust hardware measures and monitors the security daemon statically and at runtime to resist tampering.

Malicious physical access to devices is always a threat in IoT. Hardware root of trust plays an important role in defending the integrity of the IoT Edge device. Hardware root of trust come in two varieties:

- Secure elements for the protection of sensitive information like secrets and cryptographic keys.
- Secure enclaves for the protection of secrets like keys, and sensitive workloads like confidential machine learning models and metering operations.

Two kinds of execution environments exist to use hardware root of trust:

- The standard or rich execution environment (REE) that relies on the use of secure elements to protect sensitive information.
- The trusted execution environment (TEE) that relies on the use of secure enclave technology to protect sensitive information and offer protection to software execution.

For devices using secure enclaves as hardware root of trust, sensitive logic within the IoT Edge module runtime should be inside the enclave. Non-sensitive portions of the module runtime can be outside of the TEE. In all cases, we strongly recommend that original design manufacturers (ODM) and original equipment manufacturers (OEM) extend trust from their HSM to measure and defend the integrity of the IoT Edge module runtime at boot and runtime.

Minimize bloat and churn

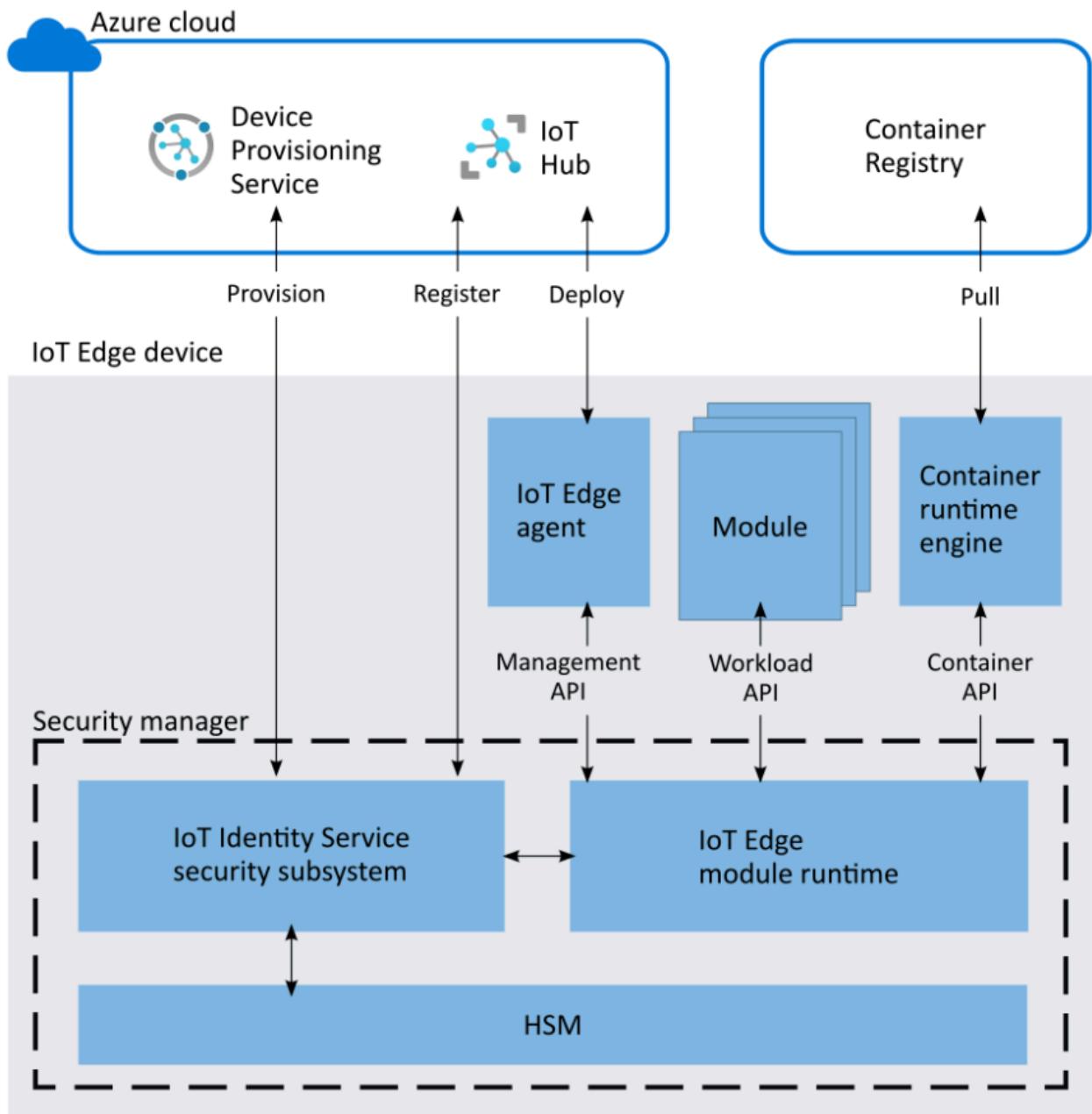
Another core principle for the IoT Edge module runtime is to minimize churn. For the highest level of trust, the IoT Edge module runtime can tightly couple with the device hardware root of trust and operate as native code. In these cases, it's common to update the IoT Edge software through the hardware root of trust's secure update paths rather than the operating system's update mechanisms, which can be challenging.

Security renewal is recommended for IoT devices, but excessive update requirements or large update payloads can expand the threat surface in many ways. For example, you may be tempted to skip some updates in order to maximize device availability. As such, the design of the IoT Edge module runtime is concise to keep the well-isolated trusted computing base small to encourage frequent updates.

Architecture

The IoT Edge module runtime takes advantage of any available hardware root of trust technology for security hardening. It also allows for split-world operation between a standard/rich execution environment (REE) and a trusted execution environment (TEE)

when hardware technologies offer trusted execution environments. Role-specific interfaces enable the major components of IoT Edge to assure the integrity of the IoT Edge device and its operations.



Cloud interface

The cloud interface enables access to cloud services that complement device security. For example, this interface allows access to the [Device Provisioning Service](#) for device identity lifecycle management.

Management API

The management API is called by the IoT Edge agent when creating/starting/stopping/removing an IoT Edge module. The module runtime stores "registrations" for all active modules. These registrations map a module's identity to

some properties of the module. For example, these module properties include the process identifier (pid) of the process running in the container and the hash of the docker container's contents.

These properties are used by the workload API to verify that the caller is authorized for an action.

The management API is a privileged API, callable only from the IoT Edge agent. Since the IoT Edge module runtime bootstraps and starts the IoT Edge agent, it verifies that the IoT Edge agent hasn't been tampered with, then it can create an implicit registration for the IoT Edge agent. The same attestation process that the workload API uses also restricts access to the management API to only the IoT Edge agent.

Container API

The container API interacts with the container system in use for module management, like Moby or Docker.

Workload API

The workload API is accessible to all modules. It provides proof of identity, either as an HSM rooted signed token or an X509 certificate, and the corresponding trust bundle to a module. The trust bundle contains CA certificates for all the other servers that the modules should trust.

The IoT Edge module runtime uses an attestation process to guard this API. When a module calls this API, the module runtime attempts to find a registration for the identity. If successful, it uses the properties of the registration to measure the module. If the result of the measurement process matches the registration, a new proof of identity is generated. The corresponding CA certificates (trust bundle) are returned to the module. The module uses this certificate to connect to IoT Hub, other modules, or start a server. When the signed token or certificate nears expiration, it's the responsibility of the module to request a new certificate.

Integration and maintenance

Microsoft maintains the main code base for the [IoT Edge module runtime](#) and the [Azure IoT identity service](#) on GitHub.

When you read the IoT Edge codebase, remember that the **module runtime** evolved from the **security daemon**. The codebase may still contain references to the security daemon.

Installation and updates

Installation and updates of the IoT Edge module runtime are managed through the operating system's package management system. IoT Edge devices with hardware root of trust should provide additional hardening to the integrity of the module runtime by managing its lifecycle through the secure boot and updates management systems. Device makers should explore these avenues based on their respective device capabilities.

Versioning

The IoT Edge runtime tracks and reports the version of the IoT Edge module runtime. The version is reported as the *runtime.platform.version* attribute of the IoT Edge agent module reported property.

Hardware security module

The IoT Edge security manager implements the Trusted Platform Module and PKCS#11 interface standards for integrating hardware security modules (HSMs). With these standards, virtually any HSM, including those with proprietary interfaces, can be integrated. We strongly recommend using HSMs for security hardening.

Secure silicon root of trust hardware

Secure silicon is necessary to anchor trust inside the IoT Edge device hardware. Secure silicon come in variety to include Trusted Platform Module (TPM), embedded Secure Element (eSE), Arm TrustZone, Intel SGX, and custom secure silicon technologies. The use of secure silicon root of trust in devices is recommended given the threats associated with physical accessibility of IoT devices.

The IoT Edge security manager aims to identify and isolate the components that defend the security and integrity of the Azure IoT Edge platform for custom hardening. Third parties, like device makers, should make use of custom security features available with their device hardware.

Learn how to harden the Azure IoT security manager with the Trusted Platform Module (TPM) using software or virtual TPMs:

Create and provision an IoT Edge device with a virtual TPM on [Linux](#) or [Linux on Windows](#).

Next steps

To learn more about securing your IoT Edge devices, read the following blog posts:

- [Securing the intelligent edge ↗](#).
- [The blueprint to securely solve the elusive zero-touch provisioning of IoT devices at scale ↗](#)
- [Solving IoT device security at scale through standards ↗](#)

Understand how Azure IoT Edge uses certificates

Article • 08/07/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

IoT Edge uses different types of certificates for different purposes. This article walks you through the different ways that IoT Edge uses certificates with Azure IoT Hub and IoT Edge gateway scenarios.

Important

For brevity, this article applies to IoT Edge version 1.2 or later. The certificate concepts for version 1.1 are similar, but there are some differences:

- The *device CA certificate* in version 1.1 was renamed to *Edge CA certificate*.
- The *workload CA certificate* in version 1.1 was retired. In version 1.2 or later, the IoT Edge module runtime generates all server certificates directly from the Edge CA certificate, without the intermediate workload CA certificate between them in the certificate chain.

Summary

These core scenarios are where IoT Edge uses certificates. Use the links to learn more about each scenario.

 Expand table

Actor	Purpose	Certificate
IoT Edge	Ensures it's communicating to the right IoT Hub	IoT Hub server certificate

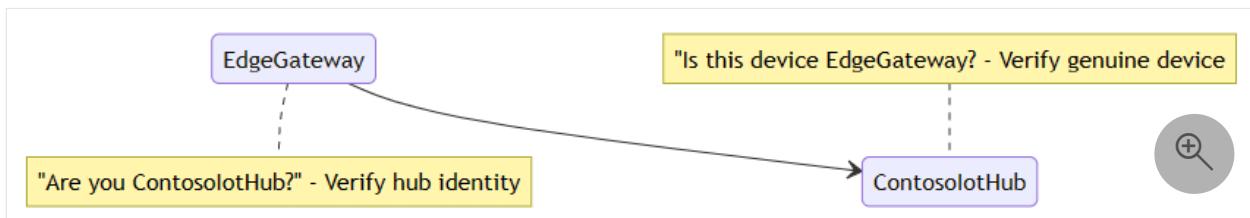
Actor	Purpose	Certificate
IoT Hub	Ensures the request came from a legitimate IoT Edge device	IoT Edge identity certificate
Downstream IoT device	Ensures it's communicating to the right IoT Edge gateway	IoT Edge Hub <i>edgeHub</i> module server certificate, issued by Edge CA
IoT Edge	Signs new module server certificates. For example, <i>edgeHub</i>	Edge CA certificate
IoT Edge	Ensures the request came from a legitimate downstream device	IoT device identity certificate

Prerequisites

- You should have a basic understanding of public key cryptography, key pairs, and how a public key and private key can encrypt or decrypt data. For more information about how IoT Edge uses public key cryptography, see [Understanding Public Key Cryptography and X.509 Public Key Infrastructure](#).
- You should have a basic understanding about how IoT Edge relates to IoT Hub. For more information, see [Understand the Azure IoT Edge runtime and its architecture](#).

Single device scenario

To help understand IoT Edge certificate concepts, imagine a scenario where an IoT Edge device named *EdgeGateway* connects to an Azure IoT Hub named *ContosolotHub*. In this example, all authentication is done with X.509 certificate authentication rather than symmetric keys. To establish trust in this scenario, we need to guarantee the IoT Hub and IoT Edge device are authentic: "*Is this device genuine and valid?*" and "*Is the identity of the IoT Hub correct?*". The scenario can be illustrated as follows:



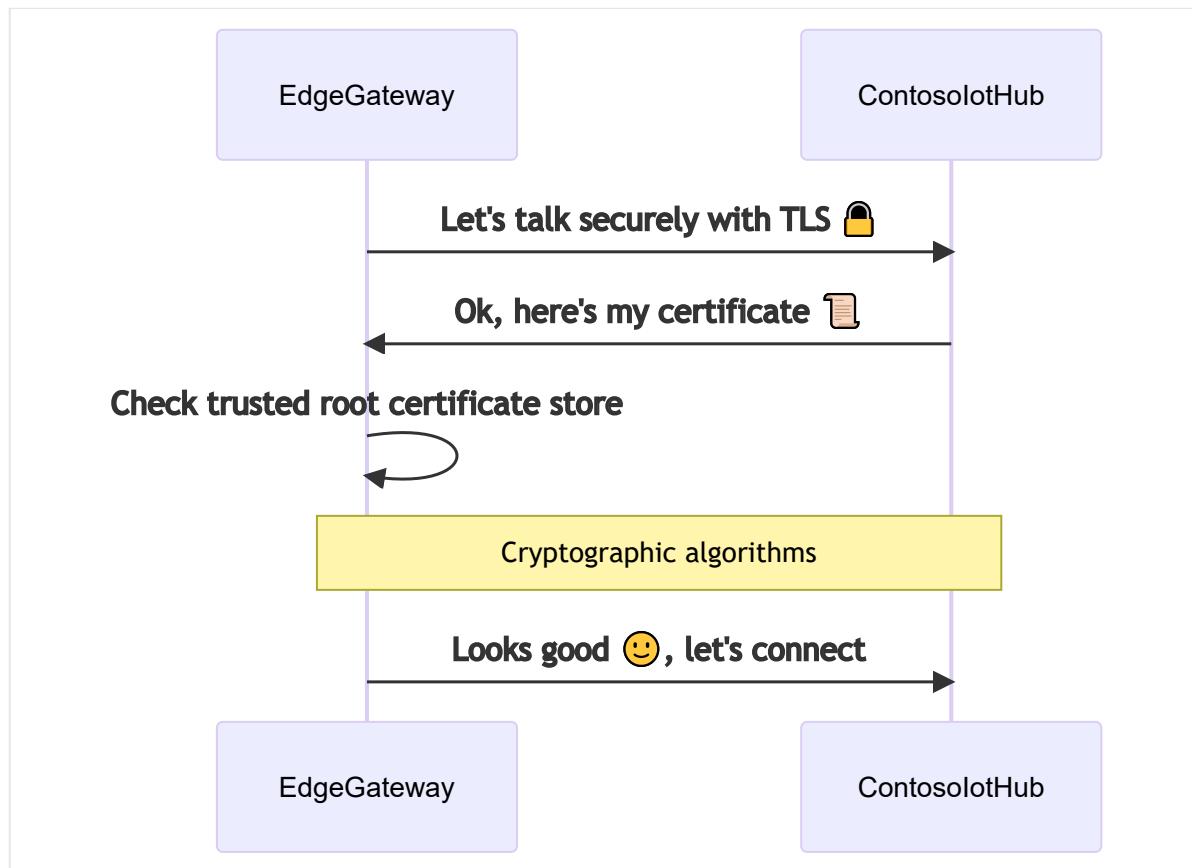
We'll explain the answers to each question and then expand the example in later sections of the article.

Device verifies IoT Hub identity

How does *EdgeGateway* verify it's communicating with the genuine *ContosolotHub*? When *EdgeGateway* wants to talk to the cloud, it connects to the endpoint *ContosolotHub.Azure-devices.NET*. To make sure the endpoint is authentic, IoT Edge needs *ContosolotHub* to show identification (ID). The ID must be issued by an authority that *EdgeGateway* trusts. To verify IoT Hub identity, IoT Edge and IoT Hub use the **TLS handshake** protocol to verify IoT Hub's server identity. A *TLS handshake* is illustrated in the following diagram. To keep the example simple, some details have been omitted. To learn more about the *TLS handshake* protocol, see [TLS handshake on Wikipedia](#).

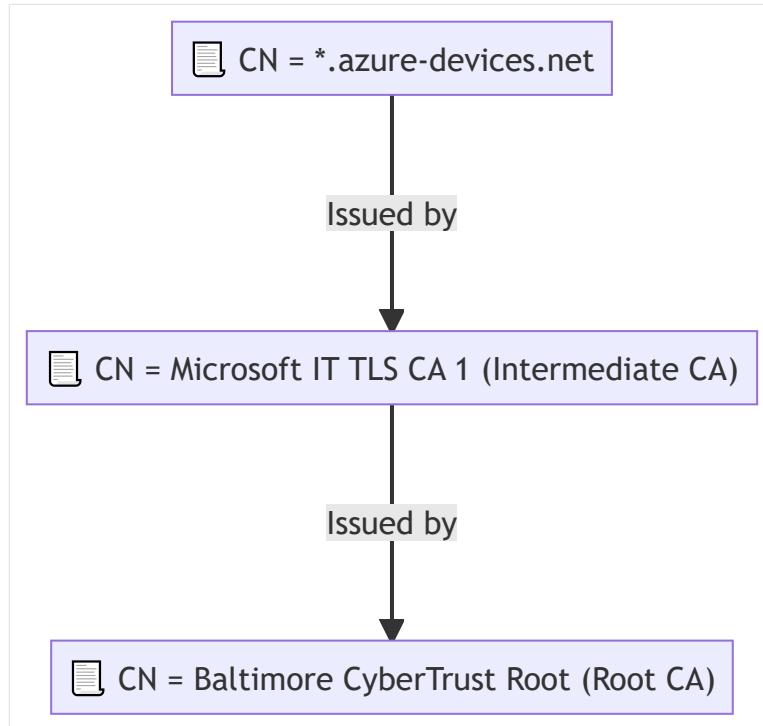
 **Note**

In this example, *ContosolotHub* represents the IoT Hub hostname *ContosolotHub.Azure-devices.NET*.



In this context, you don't need to know the exact details of the *cryptographic algorithm*. It's important to understand that the algorithm ensures the server possesses the private key that is paired with its public key. It verifies that the presenter of the certificate didn't copy or steal it. If we use a photo ID as an example, your face matches the photo on the ID. If someone steals your ID, they can't use it for identification because your face is unique and difficult to reproduce. For cryptographic keys, the key pair is related and unique. Instead of matching a face to a photo ID, the cryptographic algorithm uses the key pair to verify identity.

In our scenario, *ContosolotHub* shows the following certificate chain:



The root certificate authority (CA) is the [Baltimore CyberTrust Root](#) certificate. This root certificate is signed by DigiCert, and is widely trusted and stored in many operating systems. For example, both Ubuntu and Windows include it in the default certificate store.

Windows certificate store:

Issued To	Issued By	Expiry Date
AAA Certificate Services	AAA Certificate Services	12/2024
AddTrust External CA Root	AddTrust External CA Root	5/2024
Baltimore CyberTrust Root	Baltimore CyberTrust Root	5/2024
Certification Authority of WoSign	Certification Authority of WoSign	8/2024
Certum CA	Certum CA	6/2024
Certum Trusted Network CA	Certum Trusted Network CA	12/2024

Ubuntu certificate store:

```
stevebus@sdbsurfplap1:~$ sudo ls -l /etc/ssl/certs | grep Baltimore
lrwxrwxrwx 1 root root    29 May 21  2019 653b494a.0 -> Baltimore_CyberTrust_Root.pem
lrwxrwxrwx 1 root root    64 May 21  2019 Baltimore_CyberTrust_Root.pem -> /usr/share/ca-certificates/mozilla/Baltimore_CyberTrust_Root.crt
stevebus@sdbsurfplap1:~$
```

When a device checks for the *Baltimore CyberTrust Root* certificate, it's preinstalled in the OS. From *EdgeGateway* perspective, since the certificate chain presented by *ContosolotHub* is signed by a root CA that the OS trusts, the certificate is considered trustworthy. The certificate is known as **IoT Hub server certificate**. To learn more about the IoT Hub server certificate, see [Transport Layer Security \(TLS\) support in IoT Hub](#).

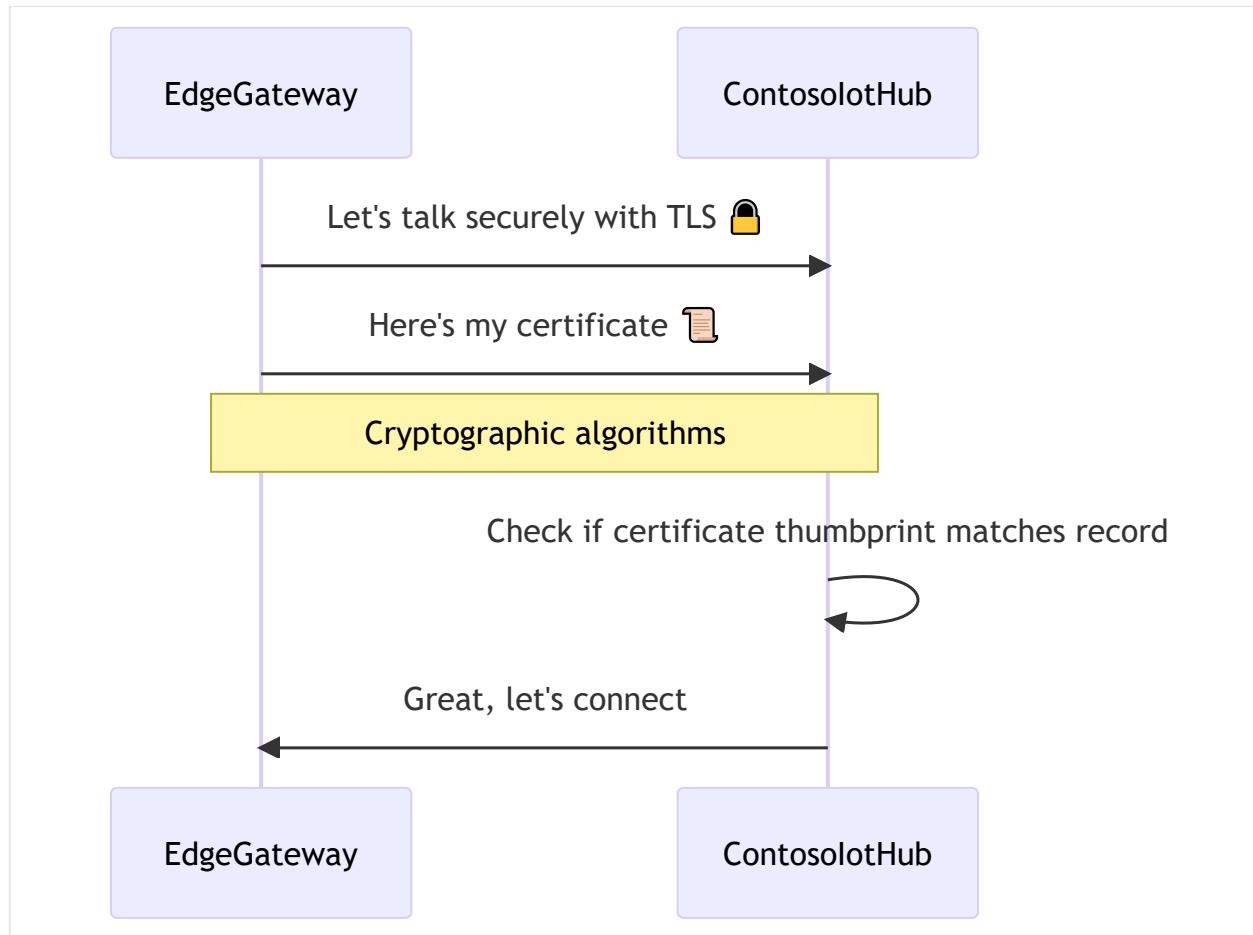
In summary, *EdgeGateway* can verify and trust *ContosolotHub*'s identity because:

- *ContosolotHub* presents its **IoT Hub server certificate**

- The server certificate is trusted in the OS certificate store
- Data encrypted with *ContosolotHub*'s public key can be decrypted by *ContosolotHub*, proving its possession of the private key

IoT Hub verifies IoT Edge device identity

How does *ContosolotHub* verify it's communicating with *EdgeGateway*? Since **IoT Hub** supports *mutual TLS* (mTLS), it checks *EdgeGateway*'s certificate during [client-authenticated TLS handshake](#). For simplicity, we'll skip some steps in the following diagram.



In this case, *EdgeGateway* provides its **IoT Edge device identity certificate**. From *ContosolotHub* perspective, it checks both that the thumbprint of the provided certificate matches its record and *EdgeGateway* has the private key paired with the certificate it presented. When you provision an IoT Edge device in IoT Hub, you provide a thumbprint. The thumbprint is what IoT Hub uses to verify the certificate.

💡 Tip

IoT Hub requires two thumbprints when registering an IoT Edge device. A best practice is to prepare two different device identity certificates with different

expiration dates. This way, if one certificate expires, the other is still valid and gives you time to rotate the expired certificate. However, it's also possible to use only one certificate for registration. Use a single certificate by setting the same certificate thumbprint for both the primary and secondary thumbprints when registering the device.

For example, we can use the following command to get the identity certificate's thumbprint on *EdgeGateway*:

Bash

```
sudo openssl x509 -in /var/lib/aziot/certd/certs/deviceid-random.cer -noout -nocert -fingerprint -sha256
```

The command outputs the certificate SHA256 thumbprint:

Output

```
SHA256
Fingerprint=1E:F3:1F:88:24:74:2C:4A:C1:A7:FA:EC:5D:16:C4:11:CD:85:52:D0:88:3
E:39:CB:7F:17:53:40:9C:02:95:C3
```

If we view the SHA256 thumbprint value for the *EdgeGateway* device registered in IoT Hub, we can see it matches the thumbprint on *EdgeGateway*:

The screenshot shows the Azure IoT Edge Device details page for a device named "EdgeGateway". The page includes sections for Device ID (EdgeGateway), Primary Thumbprint (1EF31F8824742C4AC1A7FAEC5D16C411CD8552D0883E39CB7F1753409C0295C3), Secondary Thumbprint (redacted), IoT Edge Runtime Response (200 -- OK), Enable connection to IoT Hub (Enable selected), Parent device (No parent device), and various configuration and monitoring links.

In summary, *ContosolotHub* can trust *EdgeGateway* because *EdgeGateway* presents a valid **IoT Edge device identity certificate** whose thumbprint matches the one registered in IoT Hub.

For more information about the certificate building process, see [Create and provision an IoT Edge device on Linux using X.509 certificates](#).

Note

This example doesn't address Azure IoT Hub Device Provisioning Service (DPS), which has support for X.509 CA authentication with IoT Edge when provisioned with an enrollment group. Using DPS, you upload the CA certificate or an intermediate certificate, the certificate chain is verified, then the device is provisioned. To learn more, see [DPS X.509 certificate attestation](#).

In the Azure Portal, DPS displays the SHA1 thumbprint for the certificate rather than the SHA256 thumbprint.

DPS registers or updates the SHA256 thumbprint to IoT Hub. You can verify the thumbprint using the command `openssl x509 -in /var/lib/aziot/certd/certs/deviceid-long-random-string.cer -noout -fingerprint -sha256`. Once registered, IoT Edge uses thumbprint authentication with IoT Hub. If the device is reprovisioned and a new certificate is issued, DPS updates IoT Hub with the new thumbprint.

IoT Hub currently doesn't support X.509 CA authentication directly with IoT Edge.

Certificate use for module identity operations

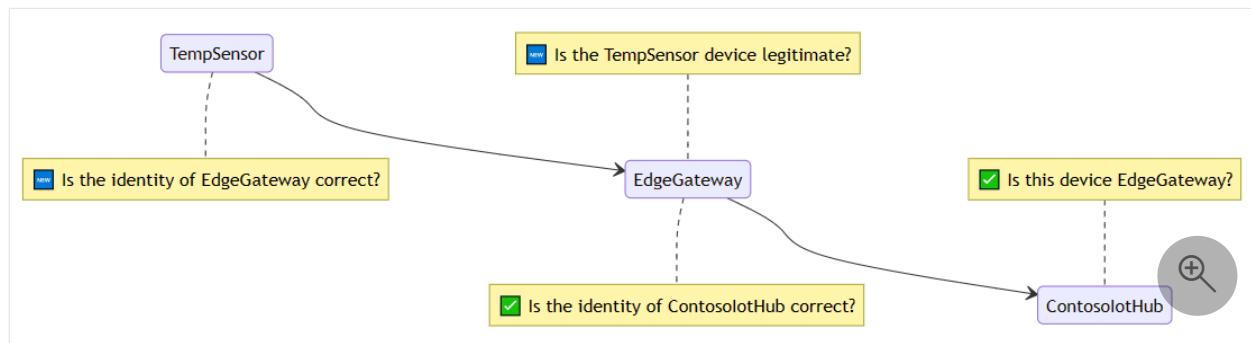
In the certificate verification diagrams, it may appear IoT Edge only uses the certificate to talk to IoT Hub. IoT Edge consists of several modules. As a result, IoT Edge uses the certificate to manage module identities for modules that send messages. The modules don't use the certificate to authenticate to IoT Hub, but rather use SAS keys derived from the private key that are generated by IoT Edge module runtime. These SAS keys don't change even if the device identity certificate expires. If the certificate expires, *edgeHub* for example continues to run, and only the module identity operations fail.

The interaction between modules and IoT Hub is secure because the SAS key is derived from a secret and IoT Edge manages the key without the risk of human intervention.

Nested device hierarchy scenario with IoT Edge as gateway

You now have a good understanding of a simple interaction IoT Edge between and IoT Hub. But, IoT Edge can also act as a gateway for downstream devices or other IoT Edge devices. These communication channels must also be encrypted and trusted. Because of the added complexity, we have to expand our example scenario to include a downstream device.

We add a regular IoT device named *TempSensor*, which connects to its parent IoT Edge device *EdgeGateway* that connects to IoT Hub *Contosolothub*. Similar to before, all authentication is done with X.509 certificate authentication. Our new scenario raises two new questions: "*Is the TempSensor device legitimate?*" and "*Is the identity of the EdgeGateway correct?*". The scenario can be illustrated as follows:

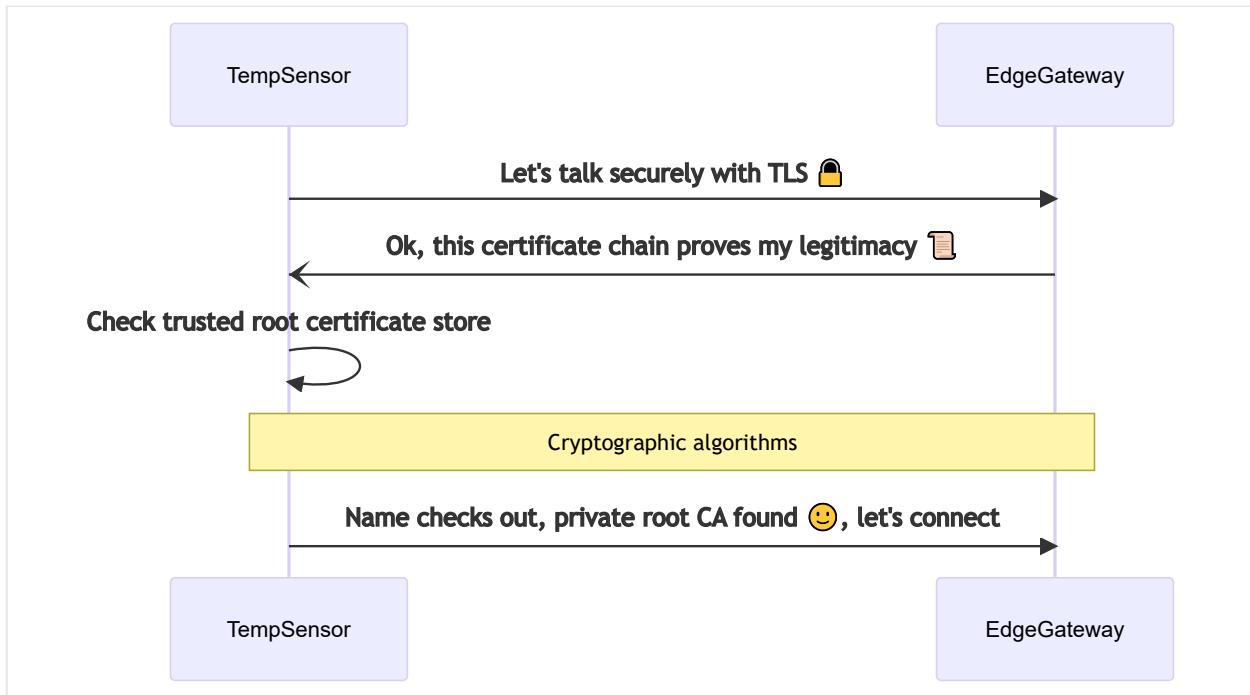


💡 Tip

TempSensor is an IoT device in the scenario. The certificate concept is the same if *TempSensor* is a downstream IoT Edge device of parent *EdgeGateway*.

Device verifies gateway identity

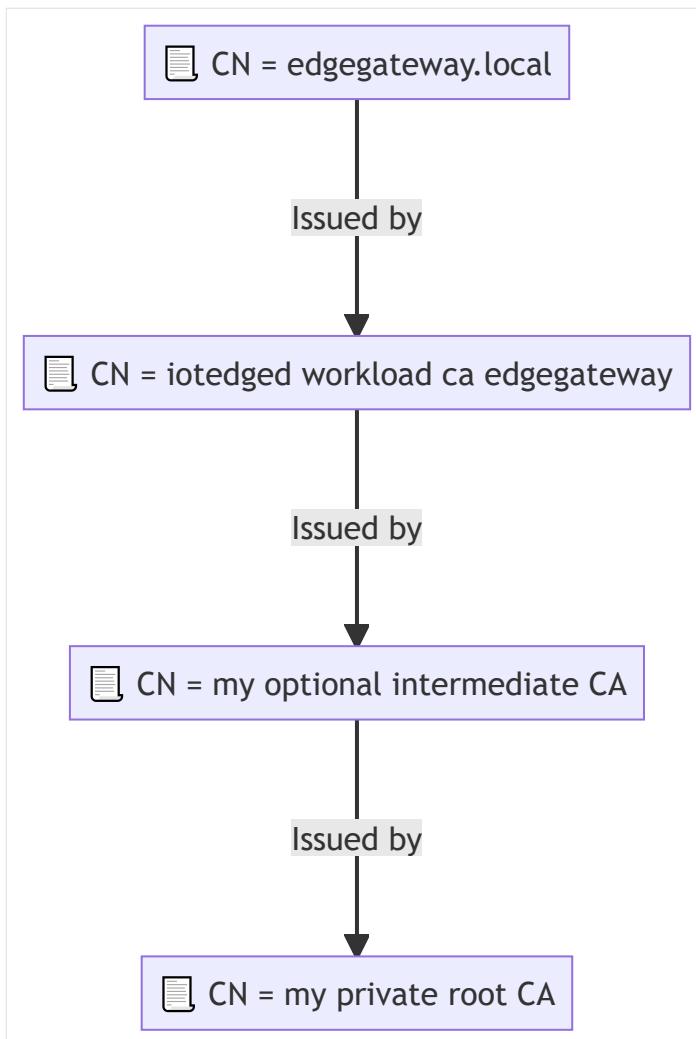
How does *TempSensor* verify it's communicating with the genuine *EdgeGateway*? When *TempSensor* wants to talk to the *EdgeGateway*, *TempSensor* needs *EdgeGateway* to show an ID. The ID must be issued by an authority that *TempSensor* trusts.



The flow is the same as when *EdgeGateway* talks to *ContosolotHub*. *TempSensor* and *EdgeGateway* use the **TLS handshake** protocol to verify *EdgeGateway*'s identity. There are two important details:

- **Hostname specificity:** The certificate presented by *EdgeGateway* must be issued to the same hostname (domain or IP address) that *TempSensor* uses to connect to *EdgeGateway*.
- **Self-signed root CA specificity:** The certificate chain presented by *EdgeGateway* is likely not in the OS default trusted root store.

To understand the details, let's first examine the certificate chain presented by *EdgeGateway*.



Hostname specificity

The certificate common name **CN = edgegateway.local** is listed at the top of the chain. **edgegateway.local** is *edgeHub*'s server certificate common name. **edgegateway.local** is also the hostname for *EdgeGateway* on the local network (LAN or VNet) where *TempSensor* and *EdgeGateway* are connected. It could be a private IP address such as 192.168.1.23 or a fully qualified domain name (FQDN) like the diagram. The *edgeHub server certificate* is generated using the **hostname** parameter defined in the [IoT Edge config.toml file](#). Don't confuse the *edgeHub server certificate* with *Edge CA certificate*. For more information about managing the Edge CA certificate, see [Manage IoT Edge certificates](#).

When *TempSensor* connects to *EdgeGateway*, *TempSensor* uses the hostname **edgegateway.local** to connect to *EdgeGateway*. *TempSensor* checks the certificate presented by *EdgeGateway* and verifies that the certificate common name is **edgegateway.local**. If the certificate common name is different, *TempSensor* rejects the connection.

ⓘ Note

For simplicity, the example shows subject certificate common name (CN) as property that is validated. In practice, if a certificate has a subject alternative name (SAN), SAN is validated instead of CN. Generally, because SAN can contain multiple values, it has both the main domain/hostname for the certificate holder as well as any alternate domains.

Why does EdgeGateway need to be told about its own hostname?

EdgeGateway doesn't have a reliable way to know how other clients on the network can connect to it. For example, on a private network, there could be DHCP servers or mDNS services that list *EdgeGateway* as `10.0.0.2` or `example-mdns-hostname.local`. But, some networks could have DNS servers that map `edgegateway.local` to *EdgeGateway*'s IP address `10.0.0.2`.

To solve the issue, IoT Edge uses the configured hostname value in `config.toml` and creates a server certificate for it. When a request comes to *edgeHub* module, it presents the certificate with the right certificate common name (CN).

Why does IoT Edge create certificates?

In the example, notice there's an *iotedged workload ca edgegateway* in the certificate chain. It's the certificate authority (CA) that exists on the IoT Edge device known as *Edge CA* (formerly known as *Device CA* in version 1.1). Like the *Baltimore CyberTrust root CA* in the earlier example, the *Edge CA* can issue other certificates. Most importantly, and also in this example, it issues the server certificate to *edgeHub* module. But, it can also issue certificates to other modules running on the IoT Edge device.

ⓘ Important

By default without configuration, *Edge CA* is automatically generated by IoT Edge module runtime when it starts for the first time, known as *quickstart Edge CA*, and then it issues a certificate to *edgeHub* module. This process speeds downstream device connection by allowing *edgeHub* to present a valid certificate that is signed. Without this feature, you'd have to get your CA to issue a certificate for *edgeHub* module. Using an automatically generated *quickstart Edge CA* isn't supported for use in production. For more information on quickstart Edge CA, see [Quickstart Edge CA](#).

Isn't it dangerous to have an issuer certificate on the device?

Edge CA is designed to enable solutions with limited, unreliable, expensive, or absent connectivity but at the same time have strict regulations or policies on certificate renewals. Without Edge CA, IoT Edge - and in particular `edgeHub` - cannot function.

To secure Edge CA in production:

- Put the EdgeCA private key in a trusted platform module (TPM), preferably in a fashion where the private key is ephemeraly generated and never leaves the TPM.
- Use a Public Key Infrastructure (PKI) to which Edge CA rolls up. This provides the ability to disable or refuse renewal of compromised certificates. The PKI can be managed by customer IT if they have the know how (lower cost) or through a commercial PKI provider.

Self-signed root CA specificity

The `edgeHub` module is an important component that makes up IoT Edge by handling all incoming traffic. In this example, it uses a certificate issued by Edge CA, which is in turn issued by a self-signed root CA. Because the root CA isn't trusted by the OS, the only way *TempSensor* would trust it is to install the CA certificate onto the device. This is also known as the *trust bundle* scenario, where you need to distribute the root to clients that need to trust the chain. The trust bundle scenario can be troublesome because you need access the device and install the certificate. Installing the certificate requires planning. It can be done with scripts, added during manufacturing, or pre-installed in the OS image.

ⓘ Note

Some clients and SDKs don't use the OS trusted root store and you need to pass the root CA file directly.

Applying all of these concepts, *TempSensor* can verify it's communicating with the genuine *EdgeGateway* because it presented a certificate that matched the address and the certificate is signed by a trusted root.

To verify the certificate chain, you could use `openssl` on the *TempSensor* device. In this example, notice that the hostname for connection matches the CN of the *depth 0* certificate, and that the root CA match.

Bash

```
openssl s_client -connect edgegateway.local:8883 --CAfile  
my_private_root_CA.pem
```

```
depth=3 CN = my_private_root_CA
verify return:1
depth=2 CN = my_optional_intermediate_CA
verify return:1
depth=1 CN = iotedged workload ca edgegateway
verify return:1
depth=0 CN = edgegateway.local
verify return: 1
CONNECTED(00000003)
---
Certificate chain
0 s:/CN=edgegateway.local
    i:/CN=iotedged workload ca edgegateway
1 s:/CN=iotedged workload ca edgegateway
    i:/CN=my_optional_intermediate_CA
2 s:/CN=my_optional_intermediate_CA
    i:/CN=my_private_root_CA
```

To learn more about `openssl` command, see [OpenSSL documentation ↗](#).

You could also inspect the certificates where they're stored by default in `/var/lib/aziot/certd/certs`. You can find *Edge CA* certificates, device identity certificates, and module certificates in the directory. You can use `openssl x509` commands to inspect the certificates. For example:

Bash

```
sudo ls -l /var/lib/aziot/certd/certs
```

Output

```
total 24
-rw-r--r-- 1 aziotcs aziotcs 1090 Jul 27 21:27 aziotedgedca-
86f154be7ff14480027f0d00c59c223db6d9e4ab0b559fc523cca36a7c973d6d.cer
-rw-r--r-- 1 aziotcs aziotcs 2589 Jun 22 18:25
aziotedgedmoduleIoTEdgeAPIProxy637913460334654299server-
c7066944a8d35ca97f1e7380ab2afea5068f39a8112476ffc89ea2c46ca81d10.cer
-rw-r--r-- 1 aziotcs aziotcs 2576 Jun 22 18:25
aziotedgedmoduleedgeHub637911101449272999server-
a0407493b6b50ee07b3fedbbb9d181e7bb5f6f52c1d071114c361aca628daa92.cer
-rw-r--r-- 1 aziotcs aziotcs 1450 Jul 27 21:27 deviceid-
bd732105ef89cf8edd2606a5309c8a26b7b5599a4e124a0fe6199b6b2f60e655.cer
```

In summary, *TempSensor* can trust *EdgeGateway* because:

- The *edgeHub* module showed a valid **IoT Edge module server certificate** for *edgegateway.local*

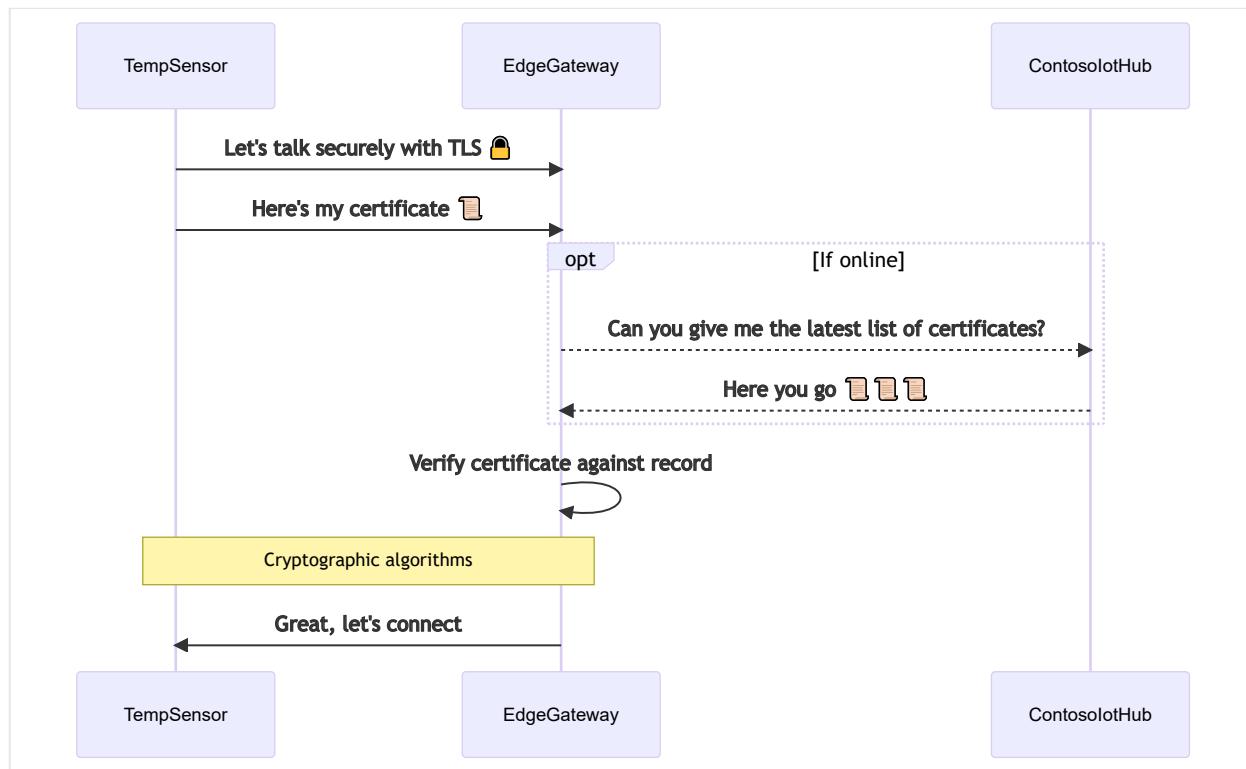
- The certificate is issued by **Edge CA** which is issued by `my_private_root_CA`
- This private root CA is also stored in the *TempSensor* as trusted root CA earlier
- Cryptographic algorithms verify that the ownership and issuance chain can be trusted

Certificates for other modules

Other modules can get server certificates issued by *Edge CA*. For example, a *Grafana* module that has a web interface. It can also get a certificate from *Edge CA*. Modules are treated as downstream devices hosted in the container. However, being able to get a certificate from the IoT Edge module runtime is a special privilege. Modules call the [workload API ↗](#) to receive the server certificate chained to the configured *Edge CA*.

Gateway verifies device identity

How does *EdgeGateway* verify it's communicating with *TempSensor*? *EdgeGateway* uses *TLS client authentication* to authenticate *TempSensor*.



The sequence is similar to *ContosolotHub* verifying a device. However, in a gateway scenario, *EdgeGateway* relies on *ContosolotHub* as the source of truth for the record of the certificates. *EdgeGateway* also keeps an offline copy or cache in case there's no connection to the cloud.

Tip

Unlike IoT Edge devices, downstream IoT devices are not limited to thumbprint X.509 authentication. X.509 CA authentication is also an option. Instead of just looking for a match on the thumbprint, *EdgeGateway* can also check if *TempSensor*'s certificate is rooted in a CA that has been uploaded to *ContosolotHub*.

In summary, *EdgeGateway* can trust *TempSensor* because:

- *TempSensor* presented a valid *IoT device identity certificate* for its name
- The identity certificate's thumbprint matches the one uploaded to *ContosolotHub*
- Cryptographic algorithms verify that the ownership and issuance chain can be trusted

Where to get the certificates and management

In most cases, you can provide your own certificates or use on auto-generated certificates. For example, *Edge CA* and the *edgeHub* certificate are auto-generated.

However, the best practice is to configure your devices to use an Enrollment over Secure Transport (EST) server to manage x509 certificates. Using an EST server frees you from manually handling the certificates and installing them on devices. For more information about using an EST server, see [Configure Enrollment over Secure Transport Server for Azure IoT Edge](#).

You can use certificates to authenticate to EST server as well. These certificates are used to authenticate with EST servers to issue other certificates. The certificate service uses a bootstrap certificate to authenticate with an EST server. The bootstrap certificate is long-lived. Upon initial authentication, the certificate service makes a request to the EST server to issue an identity certificate. This identity certificate is used in future EST requests to the same server.

If you can't use an EST server, you should request certificates from your PKI provider. You can manage the certificate files manually in IoT Hub and your IoT Edge devices. For more information, [Manage certificates on an IoT Edge device](#).

For proof of concept development, you can create test certificates. For more information, see [Create demo certificates to test IoT Edge device features](#).

Certificates in IoT

Certificate authority

The certificate authority (CA) is an entity that issues digital certificates. A certificate authority acts as a trusted third party between the owner and the receiver of the certificate. A digital certificate certifies the ownership of a public key by the receiver of the certificate. The certificate chain of trust works by initially issuing a root certificate, which is the basis for trust in all certificates issued by the authority. The root certificate owner can then issue additional intermediate certificates (downstream device certificates).

Root CA certificate

A root CA certificate is the root of trust of the entire process. In production scenarios, this CA certificate is purchased from a trusted commercial certificate authority like Baltimore, Verisign, or DigiCert. Should you have complete control over the devices connecting to your IoT Edge devices, it's possible to use a corporate level certificate authority. In either event, the entire certificate chain from the IoT Edge to IoT Hub uses it. The downstream IoT devices must trust the root certificate. You can store the root CA certificate either in the trusted root certificate authority store, or provide the certificate details in your application code.

Intermediate certificates

In a typical manufacturing process for creating secure devices, root CA certificates are rarely used directly, primarily because of the risk of leakage or exposure. The root CA certificate creates and digitally signs one or more intermediate CA certificates. There may be only one, or there may be a chain of these intermediate certificates. Scenarios that would require a chain of intermediate certificates include:

- A hierarchy of departments within a manufacturer
- Multiple companies involved serially in the production of a device
- A customer buying a root CA and deriving a signing certificate for the manufacturer to sign the devices they make on that customer's behalf

In any case, the manufacturer uses an intermediate CA certificate at the end of this chain to sign the Edge CA certificate placed on the end device. These intermediate certificates are closely guarded at the manufacturing plant. They undergo strict processes, both physical and electronic for their usage.

Next steps

- For more information about how to install certificates on an IoT Edge device and reference them from the config file, see [Manage certificate on an IoT Edge device](#).

- Understand Azure IoT Edge modules
 - Configure an IoT Edge device to act as a transparent gateway
 - This article talks about the certificates that are used to secure connections between the different components on an IoT Edge device or between an IoT Edge device and any downstream devices. You may also use certificates to authenticate your IoT Edge device to IoT Hub. Those authentication certificates are different, and aren't discussed in this article. For more information about authenticating your device with certificates, see [Create and provision an IoT Edge device using X.509 certificates](#).
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) | Get help at Microsoft Q&A

Using Private Link with IoT Edge

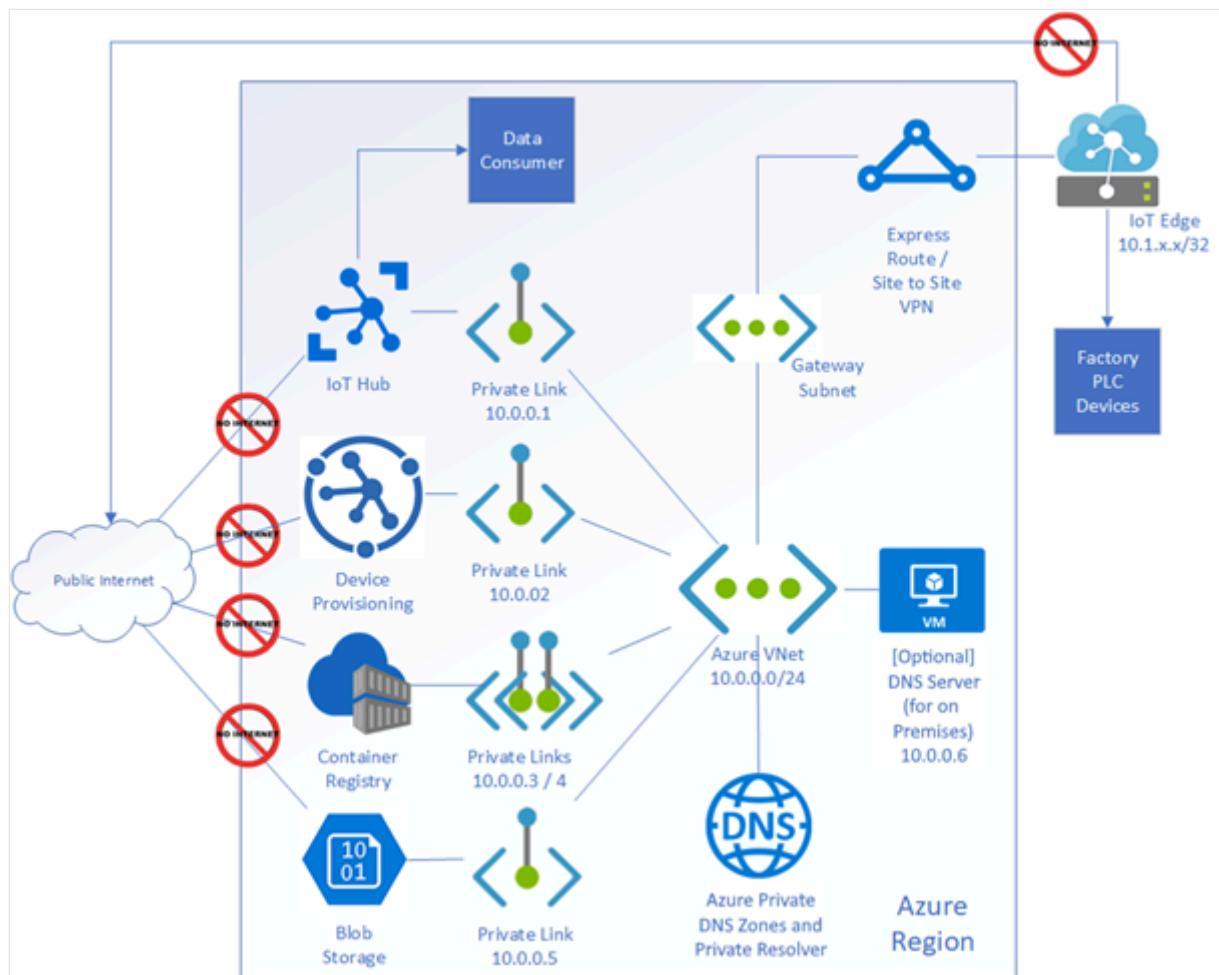
Article • 06/10/2024

Applies to: ✓ IoT Edge 1.5 ✓ IoT Edge 1.4

ⓘ Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

In Industrial IoT (IIoT) scenarios, you may want to use IoT Edge and completely isolate your network from the internet traffic. You can achieve this requirement by using various services in Azure. The following diagram is an example reference architecture for a factory network scenario.



In the preceding diagram, the network for the IoT Edge device and the PaaS services is isolated from the internet traffic. ExpressRoute or a Site-to-Site VPN facilitates an encrypted tunnel for the traffic between on premises and Azure by using Azure Private

Link service. Azure IoT services such as IoT Hub, Device Provisioning Service (DPS), Container Registry, and Blob Storage all support Private Link.

ExpressRoute

ExpressRoute lets you extend your on-premises networks into the Microsoft cloud over a private connection with the help of a connectivity provider. In IIoT, connection reliability of the devices at the edge to the cloud could be a significant requirement, and ExpressRoute fulfills this requirement via Connection Uptime SLA (Service Level Agreement). To learn more about how Azure ExpressRoute helps provide a secure connectivity for edge devices in a private network, see [What is Azure ExpressRoute?](#).

Azure Private Link

Azure Private Link enables you to access Azure PaaS services and Azure hosted customer-owned/partner services over a [private endpoint](#) in your virtual network. You can access your services running in Azure over ExpressRoute private peering, [Site-to-Site \(S2S\) VPN](#), and peered virtual networks. In IIoT, private links provide you with flexibility to connect your devices located in different regions. With private endpoint, you can also disable the access to the external PaaS resource and configure to send your traffic through the firewall. To learn more about Azure Private Link, see [What is Azure Private Link?](#).

Azure DNS Private Resolver

Azure DNS Private Resolver lets you query Azure DNS private zones from an on-premises environment and vice versa without deploying VM based DNS servers. Azure DNS Private Resolver reduces the complexity of managing both private and public IPs. The DNS forwarding ruleset feature in Azure DNS private resolver helps an IoT admin to easily configure the rules and manage the clients on what specific address an endpoint should resolve. To learn more about Azure DNS Private Resolver, see [What is Azure DNS Private Resolver?](#).

For a walk-through example scenario, see [Using Azure Private Link and Private Endpoints to secure Azure IoT traffic](#). This example illustrates a possible configuration for a factory network and not intended as a production ready reference.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback 

Confidential computing at the edge

Article • 04/09/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge supports confidential applications that run within secure enclaves on the device. Encryption provides security for data while in transit or at rest, but enclaves provide security for data and workloads while in use. IoT Edge supports Open Enclave as a standard for developing confidential applications.

Security is an important focus of the Internet of Things (IoT) because often IoT devices are often out in the world rather than secured inside a private facility. This exposure puts devices at risk for tampering and forgery because they are physically accessible to bad actors. IoT Edge devices have even more need for trust and integrity because they allow for sensitive workloads to be run at the edge. Unlike common sensors and actuators, these intelligent edge devices are potentially exposing sensitive workloads that were formerly only run within protected cloud or on-premises environments.

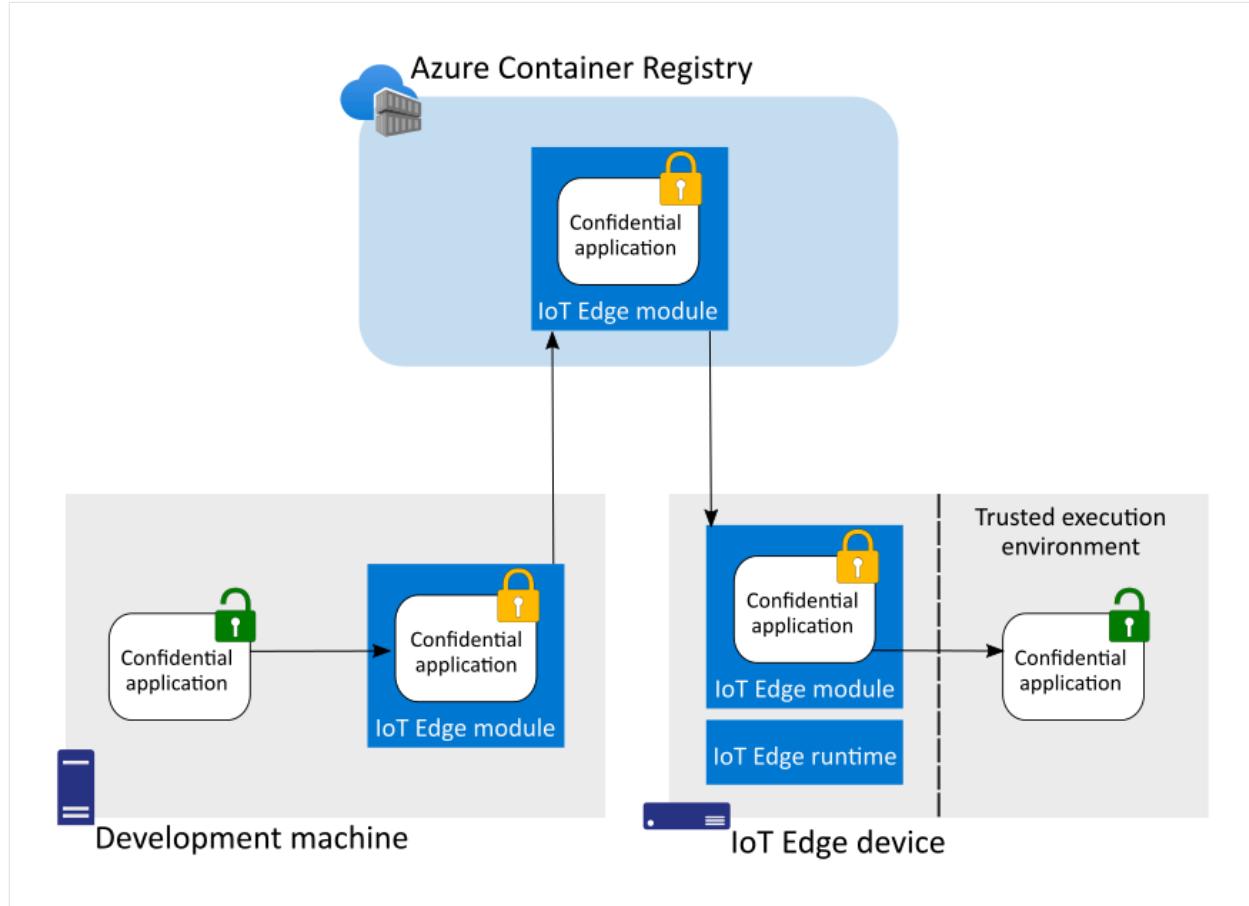
The [IoT Edge security manager](#) addresses one piece of the confidential computing challenge. The security manager uses a hardware security module (HSM) to protect the identity workloads and ongoing processes of an IoT Edge device.

Another aspect of confidential computing is protecting the data in use at the edge. A *Trusted execution environment* (TEE) is a secure, isolated environment on a processor and is sometimes referred to as an *enclave*. A *confidential application* is an application that runs in an enclave. Because of the nature of enclaves, confidential applications are protected from other apps running in the main processor or in the TEE.

Confidential applications on IoT Edge

Confidential applications are encrypted in transit and at rest, and only decrypted to run inside a trusted execution environment. This standard holds true for confidential applications deployed as IoT Edge modules.

The developer creates the confidential application and packages it as an IoT Edge module. The application is encrypted before being pushed to the container registry. The application remains encrypted throughout the IoT Edge deployment process until the module is started on the IoT Edge device. Once the confidential application is within the device's TEE, it is decrypted and can begin executing.



Confidential applications on IoT Edge are a logical extension of [Azure confidential computing](#). Workloads that run within secure enclaves in the cloud can also be deployed to run within secure enclaves at the edge.

Open Enclave

The [Open Enclave SDK](#) is an open source project that helps developers create confidential applications for multiple platforms and environments. The Open Enclave SDK operates within the trusted execution environment of a device, while the Open Enclave API acts as an interface between the TEE and the non-TEE processing environment.

Open Enclave supports multiple hardware platforms. IoT Edge support for enclaves currently requires the Open Portable TEE operating system (OP-TEE OS). To learn more, see [Open Enclave SDK for OP-TEE OS](#).

The Open Enclave repository also includes samples to help developers get started. For more information, choose one of the introductory articles:

- [Building Open Enclave SDK samples on Linux ↗](#)
- [Building Open Enclave SDK samples on Windows ↗](#)

Hardware

Currently, [TrustBox by Scalys ↗](#) is the only device supported with manufacturer service agreements for deploying confidential applications as IoT Edge modules. The TrustBox is built on The TrustBox Edge and TrustBox EdgeXL devices both come preloaded with the Open Enclave SDK and Azure IoT Edge.

For more information, see [Getting started with Open Enclave for the Scalys TrustBox ↗](#).

Develop and deploy

When you're ready to develop and deploy your confidential application, the [Microsoft Open Enclave ↗](#) extension for Visual Studio Code can help. You can use either Linux or Windows as your development machine to develop modules for the TrustBox.

Next steps

Learn how to start developing confidential applications as IoT Edge modules with the [Open Enclave extension for Visual Studio Code ↗](#).

Understand extended offline capabilities for IoT Edge devices, modules, and child devices

Article • 06/10/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge supports extended offline operations on your IoT Edge devices and enables offline operations on downstream devices too. As long as an IoT Edge device has had one opportunity to connect to IoT Hub, that device and any downstream devices can continue to function with intermittent or no internet connection.

How it works

When an IoT Edge device goes into offline mode, the IoT Edge hub takes on three roles:

- Stores any messages that would go upstream and saves them until the device reconnects.
- Acts on behalf of IoT Hub to authenticate modules and downstream devices so they can continue to operate.
- Enables communication between downstream devices that normally would go through IoT Hub.

The following example shows how an IoT Edge scenario operates in offline mode:

1. Configure devices

IoT Edge devices automatically have offline capabilities enabled. To extend that capability to other devices, you need to configure the downstream devices to trust their assigned parent device and route the device-to-cloud communications through the parent as a gateway.

2. Sync with IoT Hub

At least once after installation of the IoT Edge runtime, the IoT Edge device needs to be online to sync with IoT Hub. In this sync, the IoT Edge device gets details about any downstream devices assigned to it. The IoT Edge device also securely updates its local cache to enable offline operations and retrieves settings for local storage of telemetry messages.

3. Go offline

While disconnected from IoT Hub, the IoT Edge device, its deployed modules, and any downstream devices can operate indefinitely. Modules and downstream devices can start and restart by authenticating with the IoT Edge hub while offline. Telemetry bound upstream to IoT Hub is stored locally. Communication between modules or between downstream devices is maintained through direct methods or messages.

4. Reconnect and resync with IoT Hub

Once the connection with IoT Hub is restored, the IoT Edge device syncs again. Locally stored messages are delivered to the IoT Hub right away, but are dependent on the speed of the connection, IoT Hub latency, and related factors. They are delivered in the same order in which they were stored.

Any differences between the desired and reported properties of the modules and devices are reconciled. The IoT Edge device updates any changes to its set of assigned downstream devices.

Restrictions and limits

IoT Edge devices and their assigned downstream devices can function indefinitely offline after the initial, one-time sync. However, storage of messages depends on the [time to live \(TTL\) setting](#) and the available disk space for storing the messages.

A device's *EdgeAgent* updates its reported properties whenever there is a change in the deployment status such as a new or failed deployment. When a device is offline, the *EdgeAgent* can't report status to the Azure portal. Therefore, the device status in the Azure portal may remain **200 OK** when IoT Edge device has no internet connectivity.

Set up parent and child devices

By default, a parent device can have up to 100 children. You can change this limit by setting the **MaxConnectedClients** environment variable in the `edgeHub` module. A child device only has one parent.

Note

A downstream device emits data directly to the Internet or to gateway devices (IoT Edge-enabled or not). A child device can be a downstream device or a gateway device in a nested topology.

Downstream devices can be any device, IoT Edge or non-IoT Edge, registered to the same IoT Hub.

For more information about creating a parent-child relationship between an IoT Edge device and an IoT device, see [Authenticate a downstream device to Azure IoT Hub](#). The symmetric key, self-signed X.509, and CA-signed X.509 sections show examples of how to use the Azure portal and Azure CLI to define the parent-child relationships when creating devices. For existing devices, you can declare the relationship from the device details page in the Azure portal of either the parent or child device.

For more information about creating a parent-child relationship between two IoT Edge devices, see [Connect a downstream IoT Edge device to an Azure IoT Edge gateway](#).

Set up the parent device as a gateway

You can think of a parent/child relationship as a transparent gateway, where the child device has its own identity in IoT Hub but communicates through the cloud via its parent. For secure communication, the child device needs to be able to verify that the parent device comes from a trusted source. Otherwise, third-parties could set up malicious devices to impersonate parents and intercept communications.

One way to create this trust relationship is described in detail in the following articles:

- [Configure an IoT Edge device to act as a transparent gateway](#)
- [Connect a downstream \(child\) device to an Azure IoT Edge gateway](#)

Specify DNS servers

To improve robustness, it's highly recommended you specify the DNS server addresses used in your environment. To set your DNS server for IoT Edge, see the resolution for [Edge Agent module reports 'empty config file' and no modules start on the device](#) in the troubleshooting article.

Optional offline settings

If your devices go offline, the IoT Edge parent device stores all device-to-cloud messages until the connection is reestablished. The IoT Edge hub module manages the storage and forwarding of offline messages.

For devices that may go offline for extended periods of time, optimize performance by configuring two IoT Edge hub settings:

- Increase the *time to live* setting, so the IoT Edge hub keeps messages until your device reconnects.
- Add additional disk space for message storage.

Time to live

The *time to live* setting is the amount of time (in seconds) that a message can wait to be delivered before it expires. The default is 7200 seconds (two hours). The maximum value is only limited by the maximum value of an integer variable, which is around 2 billion.

This setting is a desired property of the IoT Edge hub, which is stored in the module twin. You can configure it in the Azure portal or directly in the deployment manifest.

```
JSON

"$edgeHub": {
    "properties.desired": {
        "schemaVersion": "1.1",
        "routes": {},
        "storeAndForwardConfiguration": {
            "timeToLiveSecs": 7200
        }
    }
}
```

Host storage for system modules

Messages and module state information are stored in the IoT Edge hub's local container filesystem by default. For improved reliability, especially when operating offline, you can also dedicate storage on the host IoT Edge device. For more information, see [Give modules access to a device's local storage](#).

Next steps

Learn more about how to set up a transparent gateway for your parent/child device connections:

- Configure an IoT Edge device to act as a transparent gateway
 - Authenticate a downstream device to Azure IoT Hub
 - Connect a downstream device to an Azure IoT Edge gateway
-

Feedback

Was this page helpful?



Yes



No

Provide product feedback ↗

Understand Azure IoT Edge limits and restrictions

Article • 06/05/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article explains the limits and restrictions when using IoT Edge.

Limits

Number of children in gateway hierarchy

Each IoT Edge parent device in gateway hierarchies can have up to 100 connected child devices by default.

However, it's important to know that each IoT Edge device in a nested topology must open a separate logical connection to the parent EdgeHub (or IoT Hub) on behalf of each connected client (device or module), plus one connection for itself. So the connections at each layer are not aggregated, but added.

For example, if there are 2 IoT Edge child devices in one layer L4, each in turn has 100 clients, then the parent IoT Edge device in the layer above L5 would have 202 total incoming connections from L4.

This limit can be changed by setting the **MaxConnectedClients** environment variable in the parent device's edgeHub module. But IoT Edge can run into issues with reporting its state in the twin reported properties if the number of clients exceeds a few hundred because of the IoT Hub twin size limit. In general, be careful when increasing the limit by changing this environment variable.

For more information, see [Create a gateway hierarchy](#).

Size of desired properties

IoT Hub enforces the following restrictions:

- An 8-kb size limit on the value of tags.
- A 32-kb size limit on both the value of `properties/desired` and `properties/reported`.

For more information, see [Module twin size](#).

Number of nested hierarchy layers

An IoT Edge device has a limit of five layers of IoT Edge devices linked as children below it.

For more information, see [Parent and child relationships](#).

Number of modules in a deployment

IoT Hub has the following restrictions for IoT Edge automatic deployments:

- 50 modules per deployment
 - This limit is superseded by the IoT Hub 32-kb module twin size limit. For more information, see [Be mindful of twin size limits when using custom modules](#).
- 100 deployments (including layered deployments per paid SKU hub)
- 10 deployments per free SKU hub

Restrictions

Certificates

IoT Edge certificates have the following restrictions:

- The common name (CN) can't be the same as the *hostname* that is used in the configuration file on the IoT Edge device.
- The name used by clients to connect to IoT Edge can't be the same as the common name used in the edge CA certificate.

For more information, see [Certificates for device security](#).

TPM attestation

When using TPM attestation with the device provisioning service, you need to use TPM 2.0.

For more information, see [TPM attestation device requirements](#).

Routing syntax

IoT Edge and IoT Hub routing syntax is almost identical. Supported query syntax:

- Message routing query based on message properties
- Message routing query based on message body

Not supported query syntax:

- Message routing query based on device twin

Restart policies

Don't use `on-unhealthy` or `on-failure` as values in modules' `restartPolicy` because they are unimplemented and won't initiate a restart. Only `never` and `always` restart policies are implemented.

The recommended way to automatically restart unhealthy IoT Edge modules is noted in [this workaround](#). Configure the `Healthcheck` property in the module's `createOptions` to handle a failed health check.

Troubleshooting logs

Accessing module logs from Azure portal could be delayed while modules are being updated.

If you view the **Troubleshoot** tab from your device in IoT Edge in the Azure portal, you may see the message "Unable to retrieve logs. The request failed with status code 504." The request times out and the **Runtime Status** might show as "Error" for all modules.

This ability to see the logs will resume in time. The reason the access is delayed is because `edgeAgent` may be busy starting modules so it can't simultaneously retrieve logs. Logs are pulled from Moby/Docker, so this process takes time, and the request can time out if `edgeAgent` is busy.

File upload

IoT Hub only supports file upload APIs for device identities, not module identities. Since IoT Edge exclusively uses modules, file upload isn't natively supported in IoT Edge.

For more information on uploading files with IoT Hub, see [Upload files with IoT Hub](#).

Edge agent environment variables

Changes made in `config.toml` to `edgeAgent` environment variables like the `hostname` aren't applied to `edgeAgent` if the container already existed. To apply these changes, remove the `edgeAgent` container using the command `sudo docker rm -f edgeAgent`. The IoT Edge daemon recreates the container and starts `edgeAgent` in about a minute.

NTLM Authentication

NTLM authentication is not supported. Proxies configured with NTLM authentication won't work.

IoT Edge has limited support for proxy authentication. Proxies configured for username and password authentication only are supported.

Next steps

For more information, see [IoT Hub other limits](#).

Create an IoT Edge device

Article • 06/03/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides an overview of the options available to you for installing and provisioning IoT Edge on your devices.

This article provides a look at all of the options for your IoT Edge solution and helps you:

- [Choose a platform](#)
- [Choose how to provision your devices](#)
- [Choose an authentication method](#)

By the end of this article, you'll have a clear picture of what platform, provisioning, and authentication options you want to use for your IoT Edge solution.

Get started

If you know what type of platform, provisioning, and authentication options you want to use to create an IoT Edge device, use the links in the following table to get started.

If you want more information about how to choose the right option for you, continue through this article to learn more.

 Expand table

	Linux containers on Linux hosts	Linux containers on Windows hosts
Manual provisioning (single device)	X.509 certificates	X.509 certificates
	Symmetric keys	Symmetric keys
Autoprovisioning (devices at scale)	X.509 certificates	X.509 certificates
	TPM	TPM

Linux containers on Linux hosts	Linux containers on Windows hosts
Symmetric keys	Symmetric keys

Terms and concepts

If you're not already familiar with IoT Edge terminology, review some key concepts:

IoT Edge runtime: The [IoT Edge runtime](#) is a collection of programs that turn a device into an IoT Edge device. Collectively, the IoT Edge runtime components enable IoT Edge devices to run your IoT Edge modules.

Provisioning: Each IoT Edge device must be provisioned. Provisioning is a two-step process. The first step is registering the device in an IoT hub, which creates a cloud identity that the device uses to establish the connection to its hub. The second step is configuring the device with its cloud identity. Provisioning can be done manually on a per-device basis, or it can be done at-scale using the [IoT Hub Device Provisioning Service](#).

Authentication: Your IoT Edge devices need to verify its identity when it connects to IoT Hub. You can choose which authentication method to use, like symmetric key passwords, certificate thumbprints, or trusted platform modules (TPMs).

Choose a platform

Platform options are referred to by the container operating system and the host operating system. The container operating system is the operating system used inside your IoT Edge runtime and module containers. The host operating system is the operating system of the device the IoT Edge runtime containers and modules are running on.

There are three platform options for your IoT Edge devices.

- **Linux containers on Linux hosts:** Run Linux-based IoT Edge containers directly on a Linux host. Throughout the IoT Edge docs, you'll also see this option referred to as **Linux** and **Linux containers** for simplicity.
- **Linux containers on Windows hosts:** Run Linux-based IoT Edge containers in a Linux virtual machine on a Windows host. Throughout the IoT Edge docs, you'll also see this option referred to as **Linux on Windows**, **IoT Edge for Linux on Windows**, and **EFLOW**.

- **Windows containers on Windows hosts:** Run Windows-based IoT Edge containers directly on a Windows host. Throughout the IoT Edge docs, you'll also see this option referred to as **Windows** and **Windows containers** for simplicity.

For the latest information about which operating systems are currently supported for production scenarios, see [Azure IoT Edge supported systems](#).

Linux containers on Linux

For Linux devices, the IoT Edge runtime is installed directly on the host device.

IoT Edge supports X64, ARM32, and ARM64 Linux devices. Microsoft provides official installation packages for a variety of operating systems.

Linux containers on Windows

IoT Edge for Linux on Windows hosts a Linux virtual machine on your Windows device. The virtual machine comes prebuilt with the IoT Edge runtime and updates are managed through Microsoft Update.

IoT Edge for Linux on Windows is the recommended way to run IoT Edge on Windows devices. To learn more, see [What is Azure IoT Edge for Linux on Windows](#).

Windows containers on Windows

IoT Edge version 1.2 or later doesn't support Windows containers. Windows containers are not supported beyond version 1.1.

Choose how to provision your devices

You can provision a single device or multiple devices at-scale, depending on the needs of your IoT Edge solution.

The options available for authenticating communications between your IoT Edge devices and your IoT hubs depend on what provisioning method you choose. You can read more about those options in the [Choose an authentication method section](#).

Single device

Single device provisioning refers to provisioning an IoT Edge device without the assistance of the [IoT Hub Device Provisioning Service](#) (DPS). You'll see single device

provisioning also referred to as **manual provisioning**.

Using single device provisioning, you'll need to manually enter provisioning information, like a connection string, on your devices. Manual provisioning is quick and easy to set up for only a few devices, but your workload will increase with the number of devices. Provisioning helps when you're considering the scalability of your solution.

Symmetric key and **X.509 self-signed** authentication methods are available for manual provisioning. You can read more about those options in the [Choose an authentication method section](#).

Devices at scale

Provisioning devices at-scale refers to provisioning one or more IoT Edge devices with the assistance of the [IoT Hub Device Provisioning Service](#). You'll see provisioning at-scale also referred to as **autoprovisioning**.

If your IoT Edge solution requires more than one device, autoprovisioning using DPS saves you the effort of manually entering provisioning information into the configuration files of each device. This automated model can be scaled to millions of IoT Edge devices.

You can secure your IoT Edge solution with the authentication method of your choice. **Symmetric key**, **X.509 certificates**, and **trusted platform module (TPM) attestation** authentication methods are available for provisioning devices at-scale. You can read more about those options in the [Choose an authentication method section](#).

To see more of the features of DPS, see the [Features section of the overview page](#).

Choose an authentication method

X.509 certificate attestation

Using X.509 certificates as an attestation mechanism is the recommended way to scale production and simplify device provisioning. Typically, X.509 certificates are arranged in a certificate chain of trust. Starting with a self-signed or trusted root certificate, each certificate in the chain signs the next lower certificate. This pattern creates a delegated chain of trust from the root certificate down through each intermediate certificate to the final downstream device certificate installed on a device.

You create two X.509 identity certificates and place them on the device. When you create a new device identity in IoT Hub, you provide thumbprints from both certificates.

When the device authenticates to IoT Hub, it presents one certificate and IoT Hub verifies that the certificate matches its thumbprint. The X.509 keys on the device should be stored in a Hardware Security Module (HSM). For example, PKCS#11 modules, ATECC, dTPM, etc.

This authentication method is more secure than symmetric keys and supports group enrollments that provide a simplified management experience for a high number of devices. This authentication method is recommended for production scenarios.

Trusted platform module (TPM) attestation

Using TPM attestation is a method for device provisioning that uses authentication features in both software and hardware. Each TPM chip uses a unique endorsement key to verify its authenticity.

TPM attestation is only available for provisioning at-scale with DPS, and only supports individual enrollments not group enrollments. Group enrollments aren't available because of the device-specific nature of TPM.

TPM 2.0 is required when you use TPM attestation with the device provisioning service.

This authentication method is more secure than symmetric keys and is recommended for production scenarios.

Symmetric keys attestation

Symmetric key attestation is a simple approach to authenticating a device. This attestation method represents a "Hello world" experience for developers who are new to device provisioning, or don't have strict security requirements.

When you create a new device identity in IoT Hub, the service creates two keys. You place one of the keys on the device, and it presents the key to IoT Hub when authenticating.

This authentication method is faster to get started but not as secure. Device provisioning using a TPM or X.509 certificates is more secure and should be used for solutions with more stringent security requirements.

Next steps

You can use the table of contents to navigate to the appropriate end-to-end guide for creating an IoT Edge device for your IoT Edge solution's platform, provisioning, and

authentication requirements.

You can also use the following links to go to the relevant article.

Linux containers on Linux hosts

Manually provision a single device:

- Provision a single Linux device using X.509 certificates
- Provision a single Linux device using symmetric keys

Provision multiple devices at-scale:

- Provision Linux devices at-scale using X.509 certificates
- Provision Linux devices at-scale using TPM attestation
- Provision Linux devices at-scale using symmetric keys

Linux containers on Windows hosts

Manually provision a single device:

- Provision a single Linux on Windows device using X.509 certificates
- Provision a single Linux on Windows device using symmetric keys

Provision multiple devices at-scale:

- Provision Linux on Windows devices at-scale using X.509 certificates
- Provision Linux on Windows devices at-scale using TPM attestation
- Provision Linux on Windows devices at-scale using symmetric keys

Configure IoT Edge device settings

Article • 06/27/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article shows settings and options for configuring the IoT Edge `/etc/aziot/config.toml` file of an IoT Edge device. IoT Edge uses the `config.toml` file to initialize settings for the device. Each of the sections of the `config.toml` file has several options. Not all options are mandatory, as they apply to specific scenarios.

A template containing all options can be found in the `config.toml.edge.template` file within the `/etc/aziot` directory on an IoT Edge device. You can copy the contents of the whole template or sections of the template into your `config.toml` file. Uncomment the sections you need. Be aware not to copy over parameters you have already defined.

If you change a device's configuration, use `sudo iotedge config apply` to apply the changes.

Global parameters

The `hostname`, `parent_hostname`, `trust_bundle_cert`, `allow_elevated_docker_permissions`, and `auto_reprovisioning_mode` parameters must be at the beginning of the configuration file before any other sections. Adding parameters before a collection of settings ensures they're applied correctly. For more information on valid syntax, see [toml.io ↗](#).

Hostname

To enable gateway discovery, every IoT Edge gateway (parent) device needs to specify a `hostname` parameter that its child devices use to find it on the local network. The `edgeHub` module also uses the `hostname` parameter to match with its server certificate. For more information, see [Why does EdgeGateway need to be told about its own hostname?](#).

ⓘ Note

When the hostname value isn't set, IoT Edge attempts to find it automatically. However, clients in the network may not be able to discover the device if it isn't set.

For **hostname**, replace **fqdn-device-name-or-ip-address** with your device name to override the default hostname of the device. The value can be a fully qualified domain name (FQDN) or an IP address. Use this setting as the gateway hostname on a IoT Edge gateway device.

```
toml  
hostname = "fqdn-device-name-or-ip-address"
```

Parent hostname

Parent hostname is used when the IoT Edge device is part of a hierarchy, otherwise known as a *nested edge*. Every downstream IoT Edge device needs to specify a **parent_hostname** parameter to identify its parent. In a hierarchical scenario where a single IoT Edge device is both a parent and a child device, it needs both parameters.

Replace **fqdn-parent-device-name-or-ip-address** with the name of your parent device. Use a hostname shorter than 64 characters, which is the character limit for a server certificate common name.

```
toml  
parent_hostname = "fqdn-parent-device-name-or-ip-address"
```

For more information about setting the **parent_hostname** parameter, see [Connect Azure IoT Edge devices together to create a hierarchy](#).

Trust bundle certificate

To provide a custom certificate authority (CA) certificate as a root of trust for IoT Edge and modules, specify a **trust_bundle_cert** configuration. Replace the parameter value with the file URI to the root CA certificate on your device.

```
toml  
trust_bundle_cert = "file:///var/aziot/certs/trust-bundle.pem"
```

For more information about the IoT Edge trust bundle, see [Manage trusted root CA](#).

Elevated Docker Permissions

Some docker capabilities can be used to gain root access. By default, the `--privileged` flag and all capabilities listed in the `CapAdd` parameter of the docker `HostConfig` are allowed.

If no modules require privileged or extra capabilities, use `allow_elevated_docker_permissions` to improve the security of the device.

```
toml  
  
allow_elevated_docker_permissions = false
```

Auto reprovisioning mode

The optional `auto_reprovisioning_mode` parameter specifies the conditions that decide when a device attempts to automatically reprovision with Device Provisioning Service. Auto provisioning mode is ignored if the device has been provisioned manually. For more information about setting DPS provisioning mode, see the [Provisioning](#) section in this article for more information.

One of the following values can be set:

[+] [Expand table](#)

Mode	Description
Dynamic	Reprovision when the device detects that it may have been moved from one IoT Hub to another. This mode is <i>the default</i> .
AlwaysOnStartup	Reprovision when the device is rebooted or a crash causes the daemons to restart.
OnErrorOnly	Never trigger device reprovisioning automatically. Device reprovisioning only occurs as fallback, if the device is unable to connect to IoT Hub during identity provisioning due to connectivity errors. This fallback behavior is implicit in Dynamic and AlwaysOnStartup modes as well.

For example:

```
toml
```

```
auto_reprovisioning_mode = "Dynamic"
```

For more information about device reprovisioning, see [IoT Hub Device reprovisioning concepts](#).

Provisioning

You can provision a single device or multiple devices at-scale, depending on the needs of your IoT Edge solution. The options available for authenticating communications between your IoT Edge devices and your IoT hubs depend on what provisioning method you choose.

You can provision with a connection string, symmetric key, X.509 certificate, identity certificate private key, or an identity certificate. DPS provisioning is included with various options. Choose one method for your provisioning. Replace the sample values with your own.

Manual provisioning with connection string

```
toml
```

```
[provisioning]
source = "manual"
connection_string = "HostName=example.azure-devices.net;DeviceId=my-
device;SharedAccessKey=<Shared access key>"
```

For more information about retrieving provisioning information, see [Create and provision an IoT Edge device on Linux using symmetric keys](#).

Manual provisioning with symmetric key

```
toml
```

```
[provisioning]
source = "manual"
iothub_hostname = "example.azure-devices.net"
device_id = "my-device"

[provisioning.authentication]
method = "sas"

device_id_pk = { value = "<Shared access key>" }      # inline key (base64),
or...
```

```
device_id_pk = { uri = "file:///var/aziot/secrets/device-id.key" }
# file URI, or...
device_id_pk = { uri = "pkcs11:slot-id=0;object=device%20id?pin-value=1234"
} # PKCS#11 URI
```

For more information about retrieving provisioning information, see [Create and provision an IoT Edge device on Linux using symmetric keys](#).

Manual provisioning with X.509 certificate

```
toml

[provisioning]
source = "manual"
iohub_hostname = "example.azure-devices.net"
device_id = "my-device"

[provisioning.authentication]
method = "x509"
```

For more information about provisioning using X.509 certificates, see [Create and provision an IoT Edge device on Linux using X.509 certificates](#).

DPS provisioning with symmetric key

```
toml

[provisioning]
source = "dps"
global_endpoint = "https://global.azure-devices-provisioning.net"
id_scope = "<DPS-ID-SCOPE>

# (Optional) Use to send a custom payload during DPS registration
payload = { uri = "file:///var/secrets/aziot/identityd/dps-additional-
data.json" }

[provisioning.attestation]
method = "symmetric_key"
registration_id = "my-device"

symmetric_key = { value = "<Device symmetric key>" } # inline key (base64),
or...
symmetric_key = { uri = "file:///var/aziot/secrets/device-id.key" }
# file URI, or...
symmetric_key = { uri = "pkcs11:slot-id=0;object=device%20id?pin-value=1234"
}
```

For more information about DPS provisioning with symmetric key, see [Create and provision IoT Edge devices at scale on Linux using symmetric key](#).

DPS provisioning with X.509 certificates

```
toml

[provisioning]
source = "dps"
global_endpoint = "https://global.azure-devices-provisioning.net/"
id_scope = "<DPS-ID-SCOPE>

# (Optional) Use to send a custom payload during DPS registration
payload = { uri = "file:///var/secrets/aziot/identityd/dps-additional-
data.json" }

[provisioning.attestation]
method = "x509"
registration_id = "my-device"

# Identity certificate private key
identity_pk = "file:///var/aziot/secrets/device-id.key.pem"           # file
URI, or...
identity_pk = "pkcs11[slot-id=0;object=device%20id?pin-value=1234]" # PKCS#11
URI

# Identity certificate
identity_cert = "file:///var/aziot/certs/device-id.pem"      # file URI,
or...
[provisioning.authentication.identity_cert]                      # dynamically
issued via...
method = "est"                                              # - EST
method = "local_ca"                                         # - a local CA
common_name = "my-device"                                    # with the given
common name, or...
subject = { L = "AQ", ST = "Antarctica", CN = "my-device" } # with the given
DN fields
```

(Optional) Enable automatic renewal of the device ID certificate

Autorenewal requires a known certificate issuance method. Set **method** to either `est` or `local_ca`.

Important

Only enable autorenewal if this device is configured for CA-based DPS enrollment. Using autorenewal for an individual enrollment causes the device to be unable to

reprovision.

toml

```
[provisioning.attestation.identity_cert.auto_renew]
rotate_key = true
threshold = "80%"
retry = "4%"
```

For more information about DPS provisioning with X.509 certificates, see [Create and provision IoT Edge devices at scale on Linux using X.509 certificates](#).

DPS provisioning with TPM (Trusted Platform Module)

toml

```
[provisioning]
source = "dps"
global_endpoint = "https://global.azure-devices-provisioning.net"
id_scope = "<DPS-ID-SCOPE>

# (Optional) Use to send a custom payload during DPS registration
payload = { uri = "file:///var/secrets/aziot/identityd/dps-additional-
data.json" }

[provisioning.attestation]
method = "tpm"
registration_id = "my-device"
```

If you use DPS provisioning with TPM, and require custom configuration, see the [TPM](#) section.

For more information, see [Create and provision IoT Edge devices at scale with a TPM on Linux](#).

Cloud Timeout and Retry Behavior

These settings control the timeout and retries for cloud operations, such as communication with Device Provisioning Service (DPS) during provisioning or IoT Hub for module identity creation.

The `cloud_timeout_sec` parameter is the deadline in seconds for a network request to cloud services. For example, an HTTP request. A response from the cloud service must be received before this deadline, or the request fails as a timeout.

The `cloud_retries` parameter controls how many times a request may be retried after the first try fails. The client always sends at least once, so the value is number of retries after the first try fails. For example, `cloud_retries = 2` means that the client makes a total of three attempts.

```
toml

cloud_timeout_sec = 10
cloud_retries = 1
```

Certificate issuance

If you configured any dynamically issued certs, choose your corresponding issuance method and replace the sample values with your own.

Cert issuance via EST

```
toml

[cert_issuance.est]
trusted_certs = ["file:///var/aziot/certs/est-id-ca.pem",]

[cert_issuance.est.auth]
username = "estuser"
password = "estpwd"
```

EST ID cert already on device

```
toml

identity_cert = "file:///var/aziot/certs/est-id.pem"

identity_pk = "file:///var/aziot/secrets/est-id.key.pem"      # file URI,
or...
identity_pk = "pkcs11:slot-id=0;object=est-id?pin-value=1234" # PKCS#11 URI
```

EST ID cert requested via EST bootstrap ID cert

Authentication with a TLS client certificate that is used once to create the initial EST ID certificate. After the first certificate issuance, an `identity_cert` and `identity_pk` are automatically created and used for future authentication and renewals. The Subject Common Name (CN) of the generated EST ID certificate is always the same as the

configured device ID under the provisioning section. These files must be readable by the users `aziotcs` and `aziotks`, respectively.

```
toml

bootstrap_identity_cert = "file:///var/aziot/certs/est-bootstrap-id.pem"

bootstrap_identity_pk = "file:///var/aziot/secrets/est-bootstrap-id.key.pem"
# file URI, or...
bootstrap_identity_pk = "pkcs11:slot-id=0;object=est-bootstrap-id?pin-
value=1234" # PKCS#11 URI

# The following parameters control the renewal of EST identity certs. These
# certs are issued by the EST server after initial authentication with the
# bootstrap cert and managed by Certificates Service.

[cert_issuance.est.identity_auto_renew]
rotate_key = true
threshold = "80%"
retry = "4%"

[cert_issuance.est.urls]
default = "https://example.org/.well-known/est"
```

Cert issuance via local CA

```
toml

[cert_issuance.local_ca]
cert = "file:///var/aziot/certs/local-ca.pem"

pk = "file:///var/aziot/secrets/local-ca.key.pem"      # file URI, or...
pk = "pkcs11:slot-id=0;object=local-ca?pin-value=1234" # PKCS#11 URI
```

TPM (Trusted Platform Module)

If you need special configuration for the TPM when using DPS TPM provisioning, use these TPM settings.

For acceptable TCTI loader strings, see section 3.5 of [TCG TSS 2.0 TPM Command Transmission Interface \(TCTI\) API Specification](#).

Setting to an empty string causes the TCTI loader library to try loading a [predefined set of TCTI modules](#) in order.

```
toml
```

```
[tpm]
tcti = "swtpm:port=2321"
```

The TPM index persists the DPS authentication key. The index is taken as an offset from the base address for persistent objects such as `0x81000000` and must lie in the range from `0x00_00_00` to `0x7F_FF_FF`. The default value is `0x00_01_00`.

```
toml

auth_key_index = "0x00_01_00"
```

Use authorization values for endorsement and owner hierarchies, if needed. By default, these values are empty strings.

```
toml

[tpm.hierarchy_authorization]
endorsement = "hello"
owner = "world"
```

PKCS#11

If you used any PKCS#11 URIs, use the following parameters and replace the values with your PKCS#11 configuration.

```
toml

[aziot_keys]
pkcs11_lib_path = "/usr/lib/libmypkcs11.so"
pkcs11_base_slot = "pkcs11:slot-id=0?pin-value=1234"
```

Default Edge Agent

When IoT Edge starts up the first time, it bootstraps a default Edge Agent module. If you need to override the parameters provided to the default Edge Agent module, use this section and replace the values with your own.

ⓘ Note

The `agent.config.createOptions` parameter is specified as a TOML inline table. This format looks like JSON but it's not JSON. For more information, see [Inline Table ↗](#)

of the TOML v1.0.0 documentation.

toml

```
[agent]
name = "edgeAgent"
type = "docker"
imagePullPolicy = "..."    # "on-create" or "never". Defaults to "on-create"

[agent.config]
image = "mcr.microsoft.com/azureiotedge-agent:1.5"
createOptions = { HostConfig = { Binds =
["/iotedge/storage:/iotedge/storage"] } }

[agent.config.auth]
serveraddress = "example.azurecr.io"
username = "username"
password = "password"

[agent.env]
RuntimeLogLevel = "debug"
UpstreamProtocol = "AmqpWs"
storageFolder = "/iotedge/storage"
```

Daemon management and workload API endpoints

If you need to override the management and workload API endpoints, use this section and replace the values with your own.

toml

```
[connect]
workload_uri = "unix:///var/run/iotedge/workload.sock"
management_uri = "unix:///var/run/iotedge/mgmt.sock"

[listen]
workload_uri = "unix:///var/run/iotedge/workload.sock"
management_uri = "unix:///var/run/iotedge/mgmt.sock"
```

Edge Agent watchdog

If you need to override the default Edge Agent watchdog settings, use this section and replace the values with your own.

```
toml
```

```
[watchdog]
max_retries = "infinite"    # the string "infinite" or a positive integer.
Defaults to "infinite"
```

Edge CA certificate

If you have your own [Edge CA](#) certificate that issues all your module certificates, use one of these sections and replace the values with your own.

Edge CA certificate loaded from a file

```
toml
```

```
[edge_ca]
cert = "file:///var/aziot/certs/edge-ca.pem"          # file URI

pk = "file:///var/aziot/secrets/edge-ca.key.pem"      # file URI, or...
pk = "pkcs11:slot-id=0;object=edge%20ca?pin-value=1234" # PKCS#11 URI
```

Edge CA certificate issued over EST

```
toml
```

```
[edge_ca]
method = "est"
```

For more information about using an EST server, see [Tutorial: Configure Enrollment over Secure Transport Server for Azure IoT Edge](#).

Optional EST configuration for issuing the Edge CA certificate

If not set, the defaults in [cert_issuance.est] are used.

```
toml
```

```
common_name = "aziot-edge CA"
expiry_days = 90
url = "https://example.org/.well-known/est"
```

```
username = "estuser"  
password = "estpwd"
```

EST ID cert already on device

```
toml  
  
identity_cert = "file:///var/aziot/certs/est-id.pem"  
  
identity_pk = "file:///var/aziot/secrets/est-id.key.pem"      # file URI,  
or...  
identity_pk = "pkcs11:slot-id=0;object=est-id?pin-value=1234" # PKCS#11 URI
```

EST ID cert requested via EST bootstrap ID cert

```
toml  
  
bootstrap_identity_cert = "file:///var/aziot/certs/est-bootstrap-id.pem"  
  
bootstrap_identity_pk = "file:///var/aziot/secrets/est-bootstrap-id.key.pem"  
# file URI, or...  
bootstrap_identity_pk = "pkcs11:slot-id=0;object=est-bootstrap-id?pin-  
value=1234" # PKCS#11 URI
```

Edge CA certificate issued from a local CA certificate

Requires [cert_issuance.local_ca] to be set.

```
toml  
  
[edge_ca]  
method = "local_ca"  
  
# Optional configuration  
common_name = "aziot-edge CA"  
expiry_days = 90
```

Edge CA quickstart certificates

If you don't have your own Edge CA certificate used to issue all module certificates, use this section and set the number of days for the lifetime of the autogenerated self-signed Edge CA certificate. Expiration defaults to 90 days.

☒ Caution

This setting is **NOT recommended for production usage**. Please configure your own Edge CA certificate in the Edge CA certificate sections.

```
toml
```

```
[edge_ca]
auto_generated_edge_ca_expiry_days = 90
```

Edge CA certificate autorenewal

This setting manages autorenewal of the Edge CA certificate. Autorenewal applies when the Edge CA is configured as *quickstart* or when the Edge CA has an issuance `method` set. Edge CA certificates loaded from files generally can't be autorenewed as the Edge runtime doesn't have enough information to renew them.

ⓘ Important

Renewal of an Edge CA requires all server certificates issued by that CA to be regenerated. This regeneration is done by restarting all modules. The time of Edge CA renewal can't be guaranteed. If random module restarts are unacceptable for your use case, disable autorenewal by not including the [edge_ca.auto_renew] section.

```
toml
```

```
[edge_ca.auto_renew]
rotate_key = true
threshold = "80%"
retry = "4%"
```

Image garbage collection

If you need to override the default image garbage collection configuration, use this section and replace the values in this section with your own.

[] Expand table

Parameter	Description
<code>enabled</code>	Runs image garbage collection
<code>cleanup_recurrence</code>	How often you want the image garbage collection to run
<code>image_age_cleanup_threshold</code>	The <i>age</i> of unused images. Images older than the threshold are removed
<code>cleanup_time</code>	24-hour HH:MM format. When the cleanup job runs

toml

```
[image_garbage_collection]
enabled = true
cleanup_recurrence = "1d"
image_age_cleanup_threshold = "7d"
cleanup_time = "00:00"
```

Moby runtime

If you need to override the default Moby runtime configuration, use this section and replace the values with your own.

toml

```
[moby_runtime]
uri = "unix:///var/run/docker.sock"
network = "azure-iot-edge"
```

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Configure an IoT Edge device to communicate through a proxy server

Article • 02/05/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

IoT Edge devices send HTTPS requests to communicate with IoT Hub. If you connected your device to a network that uses a proxy server, you need to configure the IoT Edge runtime to communicate through the server. Proxy servers can also affect individual IoT Edge modules if they make HTTP or HTTPS requests that you haven't routed through the IoT Edge hub.

This article walks through the following four steps to configure and then manage an IoT Edge device behind a proxy server:

1. [Install the IoT Edge runtime on your device](#)

The IoT Edge installation scripts pull packages and files from the internet, so your device needs to communicate through the proxy server to make those requests. For Windows devices, the installation script also provides an offline installation option.

This step is a one-time process to configure the IoT Edge device when you first set it up. You also need these same connections when you update the IoT Edge runtime.

2. [Configure IoT Edge and the container runtime on your device](#)

IoT Edge is responsible for communications with IoT Hub. The container runtime is responsible for container management, so communicates with container registries. Both of these components need to make web requests through the proxy server.

This step is a one-time process to configure the IoT Edge device when you first set it up.

3. [Configure the IoT Edge agent properties in the config file on your device](#)

The IoT Edge daemon starts the edgeAgent module initially. Then, the edgeAgent module retrieves the deployment manifest from IoT Hub and starts all the other modules. Configure the edgeAgent module environment variables manually on the device itself, so that the IoT Edge agent can make the initial connection to IoT Hub. After the initial connection, you can configure the edgeAgent module remotely.

This step is a one-time process to configure the IoT Edge device when you first set it up.

4. [For all future module deployments, set environment variables for any module communicating through the proxy](#)

Once you set up and connect an IoT Edge device to IoT Hub through the proxy server, you need to maintain the connection in all future module deployments.

This step is an ongoing process done remotely so that every new module or deployment update maintains the device's ability to communicate through the proxy server.

Know your proxy URL

Before you begin any of the steps in this article, you need to know your proxy URL.

Proxy URLs take the following format: **protocol://proxy_host:proxy_port**.

- The **protocol** is either HTTP or HTTPS. The Docker daemon can use either protocol, depending on your container registry settings, but the IoT Edge daemon and runtime containers should always use HTTP to connect to the proxy.
- The **proxy_host** is an address for the proxy server. If your proxy server requires authentication, you can provide your credentials as part of the proxy host with the following format: **user:password@proxy_host**.
- The **proxy_port** is the network port at which the proxy responds to network traffic.

Install IoT Edge through a proxy

Whether your IoT Edge device runs on Windows or Linux, you need to access the installation packages through the proxy server. Depending on your operating system, follow the steps to install the IoT Edge runtime through a proxy server.

Linux devices

If you're installing the IoT Edge runtime on a Linux device, configure the package manager to go through your proxy server to access the installation package. For example, [Set up apt-get to use a http-proxy](#). Once you configure your package manager, follow the instructions in [Install Azure IoT Edge runtime](#) as usual.

Windows devices using IoT Edge for Linux on Windows

If you're installing the IoT Edge runtime using IoT Edge for Linux on Windows, IoT Edge is installed by default on your Linux virtual machine. You're not required to install or update any other steps.

Windows devices using Windows containers

If you're installing the IoT Edge runtime on a Windows device, you need to go through the proxy server twice. The first connection downloads the installer script file, and the second connection is during the installation to download the necessary components. You can configure proxy information in Windows settings, or include your proxy information directly in the PowerShell commands.

The following steps demonstrate an example of a windows installation using the `-proxy` argument:

1. The `Invoke-WebRequest` command needs proxy information to access the installer script. Then the `Deploy-IoTEdge` command needs the proxy information to download the installation files.

PowerShell

```
. {Invoke-WebRequest -proxy <proxy URL> -useb aka.ms/iotedge-win} |  
Invoke-Expression; Deploy-IoTEdge -proxy <proxy URL>
```

2. The `Initialize-IoTEdge` command doesn't need to go through the proxy server, so the second step only requires proxy information for `Invoke-WebRequest`.

PowerShell

```
. {Invoke-WebRequest -proxy <proxy URL> -useb aka.ms/iotedge-win} |  
Invoke-Expression; Initialize-IoTEdge
```

If you have complicated credentials for the proxy server that you can't include in the URL, use the `-ProxyCredential` parameter within `-InvokeWebRequestParameters`. For example,

PowerShell

```
$proxyCredential = (Get-Credential).GetNetworkCredential()  
• {Invoke-WebRequest -proxy <proxy URL> -ProxyCredential $proxyCredential -  
useb aka.ms/iotedge-win} | Invoke-Expression;  
Deploy-IoTEdge -InvokeWebRequestParameters @{ '-Proxy' = '<proxy URL>'; '-  
ProxyCredential' = $proxyCredential }
```

For more information about proxy parameters, see [Invoke-WebRequest](#).

Configure IoT Edge and Moby

IoT Edge relies on two daemons running on the IoT Edge device. The Moby daemon makes web requests to pull container images from container registries. The IoT Edge daemon makes web requests to communicate with IoT Hub.

You must configure both the Moby and the IoT Edge daemons to use the proxy server for ongoing device functionality. This step takes place on the IoT Edge device during initial device setup.

Moby daemon

Since Moby is built on Docker, refer to the Docker documentation to configure the Moby daemon with environment variables. Most container registries (including DockerHub and Azure Container Registries) support HTTPS requests, so the parameter that you should set is **HTTPS_PROXY**. If you're pulling images from a registry that doesn't support transport layer security (TLS), then you should set the **HTTP_PROXY** parameter.

Choose the article that applies to your IoT Edge device operating system:

- [Configure Docker daemon on Linux](#) The Moby daemon on Linux devices keeps the name Docker.
- [Configure Docker daemon on Windows](#) The Moby daemon on Windows devices is called iotedge-moby. The names are different because it's possible to run both Docker Desktop and Moby in parallel on a Windows device.

IoT Edge daemon

The IoT Edge daemon is similar to the Moby daemon. Use the following steps to set an environment variable for the service, based on your operating system.

The IoT Edge daemon always uses HTTPS to send requests to IoT Hub.

Linux

Open an editor in the terminal to configure the IoT Edge daemon.

```
Bash
```

```
sudo systemctl edit aziot-edged
```

Enter the following text, replacing <proxy URL> with your proxy server address and port. Then, save and exit.

```
ini
```

```
[Service]  
Environment="https_proxy=<proxy URL>"
```

Starting in version 1.2, IoT Edge uses the IoT identity service to handle device provisioning with IoT Hub or IoT Hub Device Provisioning Service. Open an editor in the terminal to configure the IoT identity service daemon.

```
Bash
```

```
sudo systemctl edit aziot-identityd
```

Enter the following text, replacing <proxy URL> with your proxy server address and port. Then, save and exit.

```
ini
```

```
[Service]  
Environment="https_proxy=<proxy URL>"
```

Refresh the service manager to pick up the new configurations.

```
Bash
```

```
sudo systemctl daemon-reload
```

Restart the IoT Edge system services for the changes to both daemons to take effect.

```
Bash
```

```
sudo iotedge system restart
```

Verify that your environment variables and the new configuration are present.

Bash

```
systemctl show --property=Environment aziot-edged  
systemctl show --property=Environment aziot-identityd
```

Windows using IoT Edge for Linux on Windows

Sign in to your IoT Edge for Linux on Windows virtual machine:

PowerShell

[Connect-EflowVm](#)

Follow the same steps as the Linux section of this article to configure the IoT Edge daemon.

Windows using Windows containers

Open a PowerShell window as an administrator and run the following command to edit the registry with the new environment variable. Replace <proxy url> with your proxy server address and port.

PowerShell

```
reg add HKLM\SYSTEM\CurrentControlSet\Services\iotedge /v Environment /t  
REG_MULTI_SZ /d https_proxy=<proxy URL>
```

Restart IoT Edge for the changes to take effect.

PowerShell

[Restart-Service iotedge](#)

Configure the IoT Edge agent

The IoT Edge agent is the first module to start on any IoT Edge device. This module starts for the first time based on information in the IoT Edge config file. The IoT Edge agent then connects to IoT Hub to retrieve deployment manifests. The manifest declares which other modules the device should deploy.

This step takes place once on the IoT Edge device during initial device setup.

1. Open the config file on your IoT Edge device: `/etc/aziot/config.toml`. You need administrative privileges to access the configuration file. On Linux systems, use the `sudo` command before opening the file in your preferred text editor.
2. In the config file, find the `[agent]` section, which contains all the configuration information for the edgeAgent module to use on startup. Check to make sure the `[agent]` section is without comments. If the `[agent]` section is missing, add it to the `config.toml`. The IoT Edge agent definition includes an `[agent.env]` subsection where you can add environment variables.
3. Add the `https_proxy` parameter to the environment variables section, and set your proxy URL as its value.

```
toml

[agent]
name = "edgeAgent"
type = "docker"

[agent.config]
image = "mcr.microsoft.com/azureiotedge-agent:1.5"

[agent.env]
# "RuntimeLogLevel" = "debug"
# "UpstreamProtocol" = "AmqpWs"
"https_proxy" = "<proxy URL>"
```

4. The IoT Edge runtime uses AMQP by default to talk to IoT Hub. Some proxy servers block AMQP ports. If that's the case, then you also need to configure edgeAgent to use AMQP over WebSocket. Uncomment the `UpstreamProtocol` parameter.

```
toml

[agent.config]
image = "mcr.microsoft.com/azureiotedge-agent:1.5"

[agent.env]
# "RuntimeLogLevel" = "debug"
"UpstreamProtocol" = "AmqpWs"
"https_proxy" = "<proxy URL>"
```

5. Add the `https_proxy` parameter to the environment variables section, and set your proxy URL as its value.

```
toml
```

```
[agent]
name = "edgeAgent"
type = "docker"

[agent.config]
image = "mcr.microsoft.com/azureiotedge-agent:1.5"

[agent.env]
# "RuntimeLogLevel" = "debug"
# "UpstreamProtocol" = "AmqpWs"
"https_proxy" = "<proxy URL>"
```

6. The IoT Edge runtime uses AMQP by default to talk to IoT Hub. Some proxy servers block AMQP ports. If that's the case, then you also need to configure edgeAgent to use AMQP over WebSocket. Uncomment the `UpstreamProtocol` parameter.

```
toml
```

```
[agent.config]
image = "mcr.microsoft.com/azureiotedge-agent:1.5"

[agent.env]
# "RuntimeLogLevel" = "debug"
# "UpstreamProtocol" = "AmqpWs"
"https_proxy" = "<proxy URL>"
```

7. Save the changes and close the editor. Apply your latest changes.

```
Bash
```

```
sudo iotedge config apply
```

8. Verify that your proxy settings are propagated using `docker inspect edgeAgent` in the `Env` section. If not, you must recreate the container.

```
Bash
```

```
sudo docker rm -f edgeAgent
```

9. The IoT Edge runtime should recreate `edgeAgent` within a minute. Once the `edgeAgent` container is running again, use the `docker inspect edgeAgent` command to verify that the proxy settings match the configuration file.

Configure deployment manifests

Once you configure your IoT Edge device to work with your proxy server, declare the `HTTPS_PROXY` environment variable in future deployment manifests. You can edit deployment manifests either using the Azure portal wizard or by editing a deployment manifest JSON file.

Always configure the two runtime modules, `edgeAgent` and `edgeHub`, to communicate through the proxy server so they can maintain a connection with IoT Hub. If you remove the proxy information from the `edgeAgent` module, the only way to reestablish connection is by editing the config file on the device, as described in the previous section.

In addition to the `edgeAgent` and `edgeHub` modules, other modules may need the proxy configuration. Modules that need to access Azure resources besides IoT Hub, such as blob storage, must have the `HTTPS_PROXY` variable specified in the deployment manifest file.

The following procedure is applicable throughout the life of the IoT Edge device.

Azure portal

When you use the **Set modules** wizard to create deployments for IoT Edge devices, every module has an **Environment Variables** section where you can configure proxy server connections.

To configure the IoT Edge agent and IoT Edge hub modules, select **Runtime Settings** on the first step of the wizard.

Home > myIoTHub - IoT Edge > myEdgeDevice > Set modules on device: myEdgeDevice

Set modules on device: myEdgeDevice

myIoTHub

Modules Routes Review + create

Container Registry Credentials

You can specify credentials to container registries hosting module images. Listed credentials are used to retrieve modules with a matching URL. The Edge Agent will report error code 500 if it can't find a container registry setting for a module.

NAME	ADDRESS	USER NAME	PASSWORD
Name	Address	User name	Password

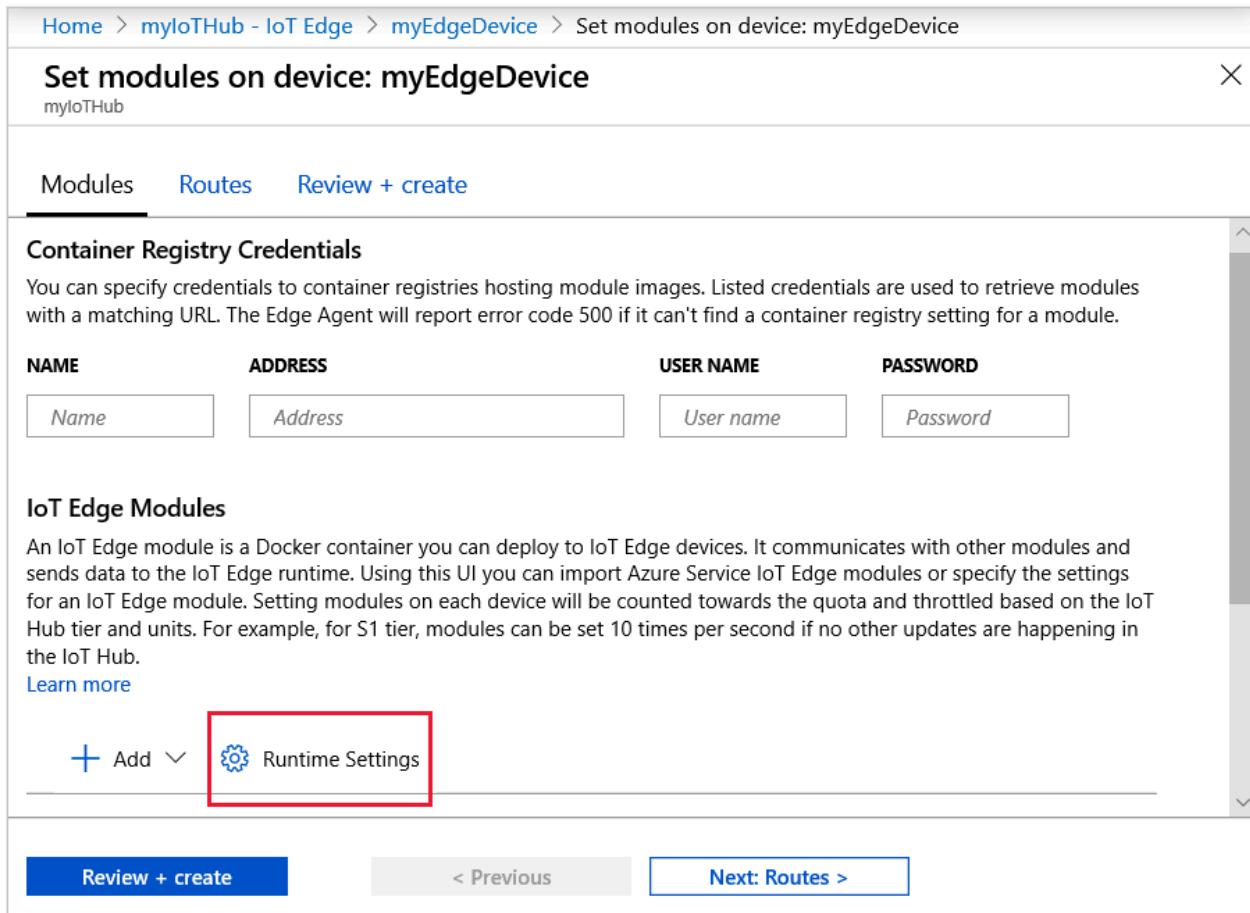
IoT Edge Modules

An IoT Edge module is a Docker container you can deploy to IoT Edge devices. It communicates with other modules and sends data to the IoT Edge runtime. Using this UI you can import Azure Service IoT Edge modules or specify the settings for an IoT Edge module. Setting modules on each device will be counted towards the quota and throttled based on the IoT Hub tier and units. For example, for S1 tier, modules can be set 10 times per second if no other updates are happening in the IoT Hub.

[Learn more](#)

+ Add ▾ Runtime Settings

[Review + create](#) < Previous [Next: Routes >](#)



Add the `https_proxy` environment variable to *both the IoT Edge agent and IoT Edge hub module* runtime settings definitions. If you included the `UpstreamProtocol` environment variable in the config file on your IoT Edge device, add that to the IoT Edge agent module definition too.

All other modules that you add to a deployment manifest follow the same pattern. Select **Apply** to save your changes.

JSON deployment manifest files

If you create deployments for IoT Edge devices using the templates in Visual Studio Code or by manually creating JSON files, you can add the environment variables directly to each module definition. If you didn't add them in the Azure portal, add them here to your JSON manifest file. Replace `<proxy URL>` with your own value.

Use the following JSON format:

```
JSON

"env": {
    "https_proxy": {
        "value": "<proxy URL>"}
```

```
    }  
}
```

With the environment variables included, your module definition should look like the following edgeHub example:

JSON

```
"edgeHub": {  
    "type": "docker",  
    "settings": {  
        "image": "mcr.microsoft.com/azureiotedge-hub:1.5",  
        "createOptions": "{}"  
    },  
    "env": {  
        "https_proxy": {  
            "value": "http://proxy.example.com:3128"  
        }  
    },  
    "status": "running",  
    "restartPolicy": "always"  
}
```

If you included the **UpstreamProtocol** environment variable in the config.yaml file on your IoT Edge device, add that to the IoT Edge agent module definition too.

JSON

```
"env": {  
    "https_proxy": {  
        "value": "<proxy URL>"  
    },  
    "UpstreamProtocol": {  
        "value": "AmqpLws"  
    }  
}
```

Working with traffic-inspecting proxies

Some proxies like [Zscaler](#) can inspect TLS-encrypted traffic. During TLS traffic inspection, the certificate returned by the proxy isn't the certificate from the target server, but instead is the certificate signed by the proxy's own root certificate. By default, IoT Edge modules (including *edgeAgent* and *edgeHub*) don't trust this proxy's certificate and the TLS handshake fails.

To resolve the failed handshake, configure both the operating system and IoT Edge modules to trust the proxy's root certificate with the following steps.

1. Configure proxy certificate in the trusted root certificate store of your host operating system. For more information about how to install a root certificate, see [Install root CA to OS certificate store](#).
2. Configure your IoT Edge device to communicate through a proxy server by referencing the certificate in the trust bundle. For more information on how to configure the trust bundle, see [Manage trusted root CA \(trust bundle\)](#).

To configure traffic inspection proxy support for containers not managed by IoT Edge, contact your proxy provider.

Fully qualified domain names (FQDNs) of destinations that IoT Edge communicates with

If your proxy's firewall requires you to add all FQDNs to your allowlist for internet connectivity, review the list from [Allow connections from IoT Edge devices](#) to determine which FQDNs to add.

Next steps

Learn more about the roles of the [IoT Edge runtime](#).

Troubleshoot installation and configuration errors with [Common issues and resolutions for Azure IoT Edge](#)

Create and provision an IoT Edge device on Linux using X.509 certificates

Article • 06/13/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides end-to-end instructions for registering and provisioning a Linux IoT Edge device, including installing IoT Edge.

Every device that connects to an IoT hub has a device ID that's used to track cloud-to-device or device-to-cloud communications. You configure a device with its connection information that includes the IoT hub hostname, the device ID, and the information the device uses to authenticate to IoT Hub.

The steps in this article walk through a process called manual provisioning, where you connect a single device to its IoT hub. For manual provisioning, you have two options for authenticating IoT Edge devices:

- **Symmetric keys:** When you create a new device identity in IoT Hub, the service creates two keys. You place one of the keys on the device, and it presents the key to IoT Hub when authenticating.

This authentication method is faster to get started, but not as secure.

- **X.509 self-signed:** You create two X.509 identity certificates and place them on the device. When you create a new device identity in IoT Hub, you provide thumbprints from both certificates. When the device authenticates to IoT Hub, it presents one certificate and IoT Hub verifies that the certificate matches its thumbprint.

This authentication method is more secure and recommended for production scenarios.

This article covers using X.509 certificates as your authentication method. If you want to use symmetric keys, see [Create and provision an IoT Edge device on Linux using](#)

[symmetric keys](#).

Note

If you have many devices to set up and don't want to manually provision each one, use one of the following articles to learn how IoT Edge works with the IoT Hub device provisioning service:

- [Create and provision IoT Edge devices at scale using X.509 certificates](#)
- [Create and provision IoT Edge devices at scale with a TPM](#)
- [Create and provision IoT Edge devices at scale using symmetric keys](#)

Prerequisites

This article covers registering your IoT Edge device and installing IoT Edge on it. These tasks have different prerequisites and utilities used to accomplish them. Make sure you have all the prerequisites covered before proceeding.

Device management tools

You can use the [Azure portal](#), [Visual Studio Code](#), or [Azure CLI](#) for the steps to register your device. Each utility has its own prerequisites or may need to be installed:

Portal

A free or standard [IoT hub](#) in your Azure subscription.

Device requirements

An X64, ARM32, or ARM64 Linux device.

Microsoft publishes installation packages for a variety of operating systems.

For the latest information about which operating systems are currently supported for production scenarios, see [Azure IoT Edge supported systems](#).

Generate device identity certificates

Manual provisioning with X.509 certificates requires IoT Edge version 1.0.10 or newer.

When you provision an IoT Edge device with X.509 certificates, you use what's called a *device identity certificate*. This certificate is only used for provisioning an IoT Edge device and authenticating the device with Azure IoT Hub. It's a leaf certificate that doesn't sign other certificates. The device identity certificate is separate from the certificate authority (CA) certificates that the IoT Edge device presents to modules or downstream devices for verification.

For X.509 certificate authentication, each device's authentication information is provided in the form of *thumbprints* taken from your device identity certificates. These thumbprints are given to IoT Hub at the time of device registration so that the service can recognize the device when it connects.

For more information about how the CA certificates are used in IoT Edge devices, see [Understand how Azure IoT Edge uses certificates](#).

You need the following files for manual provisioning with X.509:

- Two device identity certificates with their matching private key certificates in .cer or .pem formats. You need two device identity certificates for certificate rotation. A best practice is to prepare two different device identity certificates with different expiration dates. If one certificate expires, the other is still valid and gives you time to rotate the expired certificate.

One set of certificate and key files is provided to the IoT Edge runtime. When you create device identity certificates, set the certificate common name (CN) with the device ID that you want the device to have in your IoT hub.

- Thumbprints taken from both device identity certificates. IoT Hub requires two thumbprints when registering an IoT Edge device. You can use only one certificate for registration. To use a single certificate, set the same certificate thumbprint for both the primary and secondary thumbprints when registering the device.

Thumbprint values are 40-hex characters for SHA-1 hashes or 64-hex characters for SHA-256 hashes. Both thumbprints are provided to IoT Hub at the time of device registration.

One way to retrieve the thumbprint from a certificate is with the following openssl command:

Windows Command Prompt

```
openssl x509 -in <certificate filename>.pem -text -fingerprint
```

The thumbprint is included in the output of this command. For example:

Windows Command Prompt

SHA1

Fingerprint=D2:68:D9:04:9F:1A:4D:6A:FD:84:77:68:7B:C6:33:C0:32:37:51:12

If you don't have certificates available, you can [Create demo certificates to test IoT Edge device features](#). Follow the instructions in that article to set up certificate creation scripts, create a root CA certificate, and create a IoT Edge device identity certificate. For testing, you can create a single device identity certificate and use the same thumbprint for both primary and secondary thumbprint values when registering the device in IoT Hub.

Register your device

You can use the [Azure portal](#), [Visual Studio Code](#), or [Azure CLI](#) to register your device, depending on your preference.

Portal

In your IoT hub in the Azure portal, IoT Edge devices are created and managed separately from IoT devices that aren't edge enabled.

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. In the left pane, select **Devices** from the menu, then select **Add Device**.
3. On the **Create a device** page, provide the following information:
 - Create a descriptive device ID. Make a note of this device ID, as you use it later.
 - Check the **IoT Edge Device** checkbox.
 - Select **X.509 Self-Signed** as the authentication type.
 - Provide the primary and secondary identity certificate thumbprints. Thumbprint values are 40-hex characters for SHA-1 hashes or 64-hex characters for SHA-256 hashes. The Azure portal supports hexadecimal values only. Remove column separators and spaces from the thumbprint values before entering them in the portal. For example,
`D2:68:D9:04:9F:1A:4D:6A:FD:84:77:68:7B:C6:33:C0:32:37:51:12` is entered as `D268D9049F1A4D6AFD8477687BC633C032375112`.

Tip

If you are testing and want to use one certificate, you can use the same certificate for both the primary and secondary thumbprints.

4. Select **Save**.

Now that you have a device registered in IoT Hub, retrieve the information that you use to complete installation and provisioning of the IoT Edge runtime.

View registered devices and retrieve provisioning information

Devices that use X.509 certificate authentication need their IoT hub name, their device name, and their certificate files to complete installation and provisioning of the IoT Edge runtime.

Portal

The edge-enabled devices that connect to your IoT hub are listed on the **Devices** page. You can filter the list by device type *IoT Edge devices*.

Install IoT Edge

In this section, you prepare your Linux virtual machine or physical device for IoT Edge. Then, you install IoT Edge.

Run the following commands to add the package repository and then add the Microsoft package signing key to your list of trusted keys.

ⓘ Important

On June 30, 2022 Raspberry Pi OS Stretch was retired from the Tier 1 OS support list. To avoid potential security vulnerabilities update your host OS to Bullseye.

Ubuntu

Installing can be done with a few commands. Open a terminal and run the following commands:

- 22.04:

```
Bash
```

```
wget https://packages.microsoft.com/config/ubuntu/22.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

- 20.04:

```
Bash
```

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

For more information about operating system versions, see [Azure IoT Edge supported platforms](#).

 **Note**

Azure IoT Edge software packages are subject to the license terms located in each package (`usr/share/doc/{package-name}` or the `LICENSE` directory). Read the license terms prior to using a package. Your installation and use of a package constitutes your acceptance of these terms. If you don't agree with the license terms, don't use that package.

Install a container engine

Azure IoT Edge relies on an [OCI](#)-compatible container runtime. For production scenarios, we recommend that you use the Moby engine. The Moby engine is the container engine officially supported with IoT Edge. Docker CE/EE container images are compatible with the Moby runtime. If you are using Ubuntu Core snaps, the Docker snap is serviced by Canonical and supported for production scenarios.

Ubuntu

Install the Moby engine.

Bash

```
sudo apt-get update; \
  sudo apt-get install moby-engine
```

By default, the container engine doesn't set container log size limits. Over time, this can lead to the device filling up with logs and running out of disk space. However, you can configure your log to show locally, though it's optional. To learn more about logging configuration, see [Production Deployment Checklist](#).

The following steps show you how to configure your container to use [local logging driver](#) as the logging mechanism.

Ubuntu / Debian / RHEL

1. Create or edit the existing Docker [daemon's config file](#)

Bash

```
sudo nano /etc/docker/daemon.json
```

2. Set the default logging driver to the `local` logging driver as shown in the example.

JSON

```
{  
  "log-driver": "local"  
}
```

3. Restart the container engine for the changes to take effect.

Bash

```
sudo systemctl restart docker
```

Install the IoT Edge runtime

The IoT Edge service provides and maintains security standards on the IoT Edge device. The service starts on every boot and bootstraps the device by starting the rest of the IoT

Edge runtime.

Note

Beginning with version 1.2, the [IoT identity service](#) handles identity provisioning and management for IoT Edge and for other device components that need to communicate with IoT Hub.

The steps in this section represent the typical process to install the latest IoT Edge version on a device that has internet connection. If you need to install a specific version, like a pre-release version, or need to install while offline, follow the [Offline or specific version installation](#) steps later in this article.

Tip

If you already have an IoT Edge device running an older version and want to upgrade to the latest release, use the steps in [Update the IoT Edge security daemon and runtime](#). Later versions are sufficiently different from previous versions of IoT Edge that specific steps are necessary to upgrade.

Ubuntu

Install the latest version of IoT Edge and the IoT identity service package (if you're not already [up-to-date](#)):

- 22.04:

Bash

```
sudo apt-get update; \
    sudo apt-get install aziot-edge
```

- 20.04:

Bash

```
sudo apt-get update; \
    sudo apt-get install aziot-edge defender-iot-micro-agent-edge
```

The optional `defender-iot-micro-agent-edge` package includes the Microsoft Defender for IoT security micro-agent that provides endpoint visibility into security

posture management, vulnerabilities, threat detection, fleet management and more to help you secure your IoT Edge devices. It's recommended to install the micro agent with the Edge agent to enable security monitoring and hardening of your Edge devices. To learn more about Microsoft Defender for IoT, see [What is Microsoft Defender for IoT for device builders](#).

Provision the device with its cloud identity

Now that the container engine and the IoT Edge runtime are installed on your device, you're ready to set up the device with its cloud identity and authentication information.

Ubuntu / Debian / RHEL

1. Create the configuration file for your device based on a template file that's provided as part of the IoT Edge installation.

Bash

```
sudo cp /etc/aziot/config.toml.edge.template /etc/aziot/config.toml
```

2. On the IoT Edge device, open the configuration file.

Bash

```
sudo nano /etc/aziot/config.toml
```

3. Find the **Provisioning** section of the file and uncomment the lines for manual provisioning with X.509 identity certificate. Make sure that any other provisioning sections are commented out.

toml

```
# Manual provisioning with x.509 certificates
[provisioning]
source = "manual"
iothub_hostname = "REQUIRED_IOTHUB_HOSTNAME"
device_id = "REQUIRED_DEVICE_ID_PROVISIONED_IN_IOTHUB"

[provisioning.authentication]
method = "x509"

identity_cert =
"REQUIRED_URI_OR_POINTER_TO_DEVICE_IDENTITY_CERTIFICATE"
```

```
identity_pk = "REQUIRED_URI_TO_DEVICE_IDENTITY_PRIVATE_KEY"
```

Update the following fields:

- **iothub_hostname**: Hostname of the IoT Hub the device connects to. For example, `{IoT hub name}.azure-devices.net`.
- **device_id**: The ID that you provided when you registered the device.
- **identity_cert**: URI to an identity certificate on the device, for example: `file:///path/identity_certificate.pem`. Or, dynamically issue the certificate using EST or a local certificate authority.
- **identity_pk**: URI to the private key file for the provided identity certificate, for example: `file:///path/identity_key.pem`. Or, provide a PKCS#11 URI and then provide your configuration information in the

PKCS#11 section later in the config file.

For more information about certificates, see [Manage IoT Edge certificates](#).

Save and close the file.

```
CTRL + X, Y, Enter
```

After entering the provisioning information in the configuration file, apply your changes:

```
Bash
```

```
sudo iotedge config apply
```

Deploy modules

To deploy your IoT Edge modules, go to your IoT hub in the Azure portal, then:

1. Select **Devices** from the IoT Hub menu.
2. Select your device to open its page.
3. Select the **Set Modules** tab.
4. Since we want to deploy the IoT Edge default modules (`edgeAgent` and `edgeHub`), we don't need to add any modules to this pane, so select **Review + create** at the bottom.

5. You see the JSON confirmation of your modules. Select **Create** to deploy the modules.<

For more information, see [Deploy a module](#).

Verify successful configuration

Verify that the runtime was successfully installed and configured on your IoT Edge device.

Tip

You need elevated privileges to run `iotedge` commands. Once you sign out of your machine and sign back in the first time after installing the IoT Edge runtime, your permissions are automatically updated. Until then, use `sudo` in front of the commands.

Check to see that the IoT Edge system service is running.

```
Bash
```

```
sudo iotedge system status
```

A successful status response is `Ok`.

If you need to troubleshoot the service, retrieve the service logs.

```
Bash
```

```
sudo iotedge system logs
```

Use the `check` tool to verify configuration and connection status of the device.

```
Bash
```

```
sudo iotedge check
```

You can expect a range of responses that may include **OK** (green), **Warning** (yellow), or **Error** (red). For troubleshooting common errors, see [Solutions to common issues for Azure IoT Edge](#).

```
Configuration checks
-----
✓ aziot-edged configuration is well-formed - OK
✓ configuration up-to-date with config.toml - OK
✓ container engine is installed and functional - OK
✓ configuration has correct URIs for daemon mgmt endpoint - OK
✓ aziot-edge package is up-to-date - OK
✓ container time is close to host time - OK
☒ DNS server - Warning
    Container engine is not configured with DNS server setting, which may impact connectivity to IoT Hub.
    Please see https://aka.ms/iotedge-prod-checklist-dns for best practices.
    You can ignore this warning if you are setting DNS server per module in the Edge deployment.
!! production readiness: logs policy - Warning
    Container engine is not configured to rotate module logs which may cause it run out of disk space.
    Please see https://aka.ms/iotedge-prod-checklist-logs for best practices.
    You can ignore this warning if you are setting log policy per module in the Edge deployment.
✗ production readiness: Edge Agent's storage directory is persisted on the host filesystem - Error
    Could not check current state of edgeAgent container
✗ production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error
    Could not check current state of edgeHub container
✓ Agent image is valid and can be pulled from upstream - OK
✓ proxy settings are consistent in aziot-edged, aziot-identityd, moby daemon and config.toml - OK
```

💡 Tip

Always use `sudo` to run the check tool, even after your permissions are updated. The tool needs elevated privileges to access the config file to verify configuration status.

❗ Note

On a newly provisioned device, you may see an error related to IoT Edge Hub:

✗ **production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error**

Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module isn't running. To resolve the error, in IoT Hub, set the modules for the device and create a deployment. Creating a deployment for the device starts the modules on the device including the IoT Edge Hub module.

View all the modules running on your IoT Edge device. When the service starts for the first time, you should only see the **edgeAgent** module running. The edgeAgent module runs by default and helps to install and start any additional modules that you deploy to your device.

Bash

```
sudo iotedge list
```

When you create a new IoT Edge device, it displays the status code 417 -- The device's deployment configuration is not set in the Azure portal. This status is normal, and means that the device is ready to receive a module deployment.

Offline or specific version installation (optional)

The steps in this section are for scenarios not covered by the standard installation steps. This may include:

- Install IoT Edge while offline
- Install a release candidate version

Use the steps in this section if you want to install a specific version of the Azure IoT Edge runtime that isn't available through your package manager. The Microsoft package list only contains a limited set of recent versions and their sub-versions, so these steps are for anyone who wants to install an older version or a release candidate version.

If you're using Ubuntu snaps, you can download a snap and install it offline. For more information, see [Download snaps and install offline](#).

Using curl commands, you can target the component files directly from the IoT Edge GitHub repository.

1. Navigate to the [Azure IoT Edge releases](#), and find the release version that you want to target.
2. Expand the **Assets** section for that version.
3. Every release should have new files for IoT Edge and the identity service. If you're going to install IoT Edge on an offline device, download these files ahead of time. Otherwise, use the following commands to update those components.
 - a. Find the **aziot-identity-service** file that matches your IoT Edge device's architecture. Right-click on the file link and copy the link address.
 - b. Use the copied link in the following command to install that version of the identity service:

```
Ubuntu / Debian

Bash

curl -L <identity service link> -o aziot-identity-service.deb &&
```

```
sudo apt-get install ./aziot-identity-service.deb
```

c. Find the **aziot-edge** file that matches your IoT Edge device's architecture. Right-click on the file link and copy the link address.

d. Use the copied link in the following command to install that version of IoT Edge.

Ubuntu / Debian

Bash

```
curl -L <iotedge link> -o aziot-edge.deb && sudo apt-get install  
./aziot-edge.deb
```

Uninstall IoT Edge

If you want to remove the IoT Edge installation from your device, use the following commands.

Remove the IoT Edge runtime.

Ubuntu / Debian

Bash

```
sudo apt-get autoremove --purge aziot-edge
```

Leave out the `--purge` flag if you plan to reinstall IoT Edge and use the same configuration information in the future. The `--purge` flags delete all the files associated with IoT Edge, including your configuration files.

When the IoT Edge runtime is removed, any containers that it created are stopped but still exist on your device. View all containers to see which ones remain.

Bash

```
sudo docker ps -a
```

Delete the containers from your device, including the two runtime containers.

Bash

```
sudo docker rm -f <container name>
```

Finally, remove the container runtime from your device.

Ubuntu / Debian

Bash

```
sudo apt-get autoremove --purge moby-engine
```

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Create and provision an IoT Edge device on Linux using symmetric keys

Article • 03/13/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides end-to-end instructions for registering and provisioning a Linux IoT Edge device that includes installing IoT Edge.

Each device that connects to an [IoT hub](#) has a device ID that's used to track [cloud-to-device](#) or [device-to-cloud](#) communications. You configure a device with its connection information, which includes:

- IoT hub hostname
- Device ID
- Authentication details to connect to IoT Hub

The steps in this article walk through a process called *manual provisioning*, where you connect a single device to its IoT hub. For manual provisioning, you have two options for authenticating IoT Edge devices:

- **Symmetric keys:** When you create a new device identity in IoT Hub, the service creates two keys. You place one of the keys on the device, and it presents the key to IoT Hub when authenticating.

This authentication method is faster to get started, but not as secure.

- **X.509 self-signed:** You create two X.509 identity certificates and place them on the device. When you create a new device identity in IoT Hub, you provide thumbprints from both certificates. When the device authenticates to IoT Hub, it presents one certificate and IoT Hub verifies that the certificate matches its thumbprint.

This authentication method is more secure and recommended for production scenarios.

This article covers using symmetric keys as your authentication method. If you want to use X.509 certificates, see [Create and provision an IoT Edge device on Linux using X.509 certificates](#).

Note

If you have many devices to set up and don't want to manually provision each one, use one of the following articles to learn how IoT Edge works with the IoT Hub device provisioning service:

- [Create and provision IoT Edge devices at scale using X.509 certificates](#)
- [Create and provision IoT Edge devices at scale with a TPM](#)
- [Create and provision IoT Edge devices at scale using symmetric keys](#)

Prerequisites

This article shows how to register your IoT Edge device and install IoT Edge (also called IoT Edge runtime) on your device. Make sure you have the device management tool of your choice, for example Azure CLI, and device requirements before you register and install your device.

Device management tools

You can use the [Azure portal](#), [Visual Studio Code](#), or [Azure CLI](#) for the steps to register your device. Each utility has its own prerequisites or may need to be installed:

Portal

A free or standard [IoT hub](#) in your Azure subscription.

Device requirements

An X64, ARM32, or ARM64 Linux device.

Microsoft publishes installation packages for a variety of operating systems.

For the latest information about which operating systems are currently supported for production scenarios, see [Azure IoT Edge supported systems](#).

Visual Studio Code extensions

If you are using Visual Studio Code, there are helpful Azure IoT extensions that make the device creation and management process easier.

Install both the Azure IoT Edge and Azure IoT Hub extensions:

- [Azure IoT Edge](#). The *Azure IoT Edge tools for Visual Studio Code* extension is in maintenance mode.
- [Azure IoT Hub](#)

Register your device

You can use the **Azure portal**, **Visual Studio Code**, or **Azure CLI** to register your device, depending on your preference.

Portal

In your IoT hub in the Azure portal, IoT Edge devices are created and managed separately from IoT devices that are not edge enabled.

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. In the left pane, select **Devices** from the menu, then select **Add Device**.
3. On the **Create a device** page, provide the following information:
 - Create a descriptive Device ID, for example `my-edge-device-1` (all lowercase). Copy this Device ID, as you'll use it later.
 - Check the **IoT Edge Device** checkbox.
 - Select **Symmetric key** as the authentication type.
 - Use the default settings to auto-generate authentication keys, which connect the new device to your hub.
4. Select **Save**.

You should see your new device listed in your IoT hub.

Now that you have a device registered in IoT Hub, you can retrieve provisioning information used to complete the installation and provisioning of the [IoT Edge runtime](#) in the next step.

View registered devices and retrieve provisioning information

Devices that use symmetric key authentication need their connection strings to complete installation and provisioning of the IoT Edge runtime. The connection string gets generated for your IoT Edge device when you create the device. For Visual Studio Code and Azure CLI, the connection string is in the JSON output. If you use the Azure portal to create your device, you can find the connection string from the device itself. When you select your device in your IoT hub, it's listed as **Primary connection string** on the device page.

Portal

The edge-enabled devices that connect to your IoT hub are listed on the **Devices** page of your IoT hub. If you have multiple devices, you can filter the list by selecting the type **IoT Edge Devices**, then select **Apply**.

When you're ready to set up your device, you need the connection string that links your physical device with its identity in the IoT hub. Devices that authenticate with symmetric keys have their connection strings available to copy in the portal. To find your connection string in the portal:

1. From the **Devices** page, select the IoT Edge device ID from the list.
2. Copy the value of either **Primary Connection String** or **Secondary Connection String**. Either key will work.

Install IoT Edge

In this section, you prepare your Linux virtual machine or physical device for IoT Edge. Then, you install IoT Edge.

Run the following commands to add the package repository and then add the Microsoft package signing key to your list of trusted keys.

ⓘ Important

On June 30, 2022 Raspberry Pi OS Stretch was retired from the Tier 1 OS support list. To avoid potential security vulnerabilities update your host OS to Bullseye.

Installing can be done with a few commands. Open a terminal and run the following commands:

- 22.04:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/22.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

- 20.04:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

For more information about operating system versions, see [Azure IoT Edge supported platforms](#).

 **Note**

Azure IoT Edge software packages are subject to the license terms located in each package (`usr/share/doc/{package-name}` or the `LICENSE` directory). Read the license terms prior to using a package. Your installation and use of a package constitutes your acceptance of these terms. If you don't agree with the license terms, don't use that package.

Install a container engine

Azure IoT Edge relies on an [OCI](#)-compatible container runtime. For production scenarios, we recommend that you use the Moby engine. The Moby engine is the only container engine officially supported with IoT Edge. Docker CE/EE container images are compatible with the Moby runtime.

Ubuntu

Install the Moby engine.

Bash

```
sudo apt-get update; \
  sudo apt-get install moby-engine
```

By default, the container engine doesn't set container log size limits. Over time, this can lead to the device filling up with logs and running out of disk space. However, you can configure your log to show locally, though it's optional. To learn more about logging configuration, see [Production Deployment Checklist](#).

The following steps show you how to configure your container to use [local logging driver](#) as the logging mechanism.

Ubuntu / Debian / RHEL

1. Create or edit the existing Docker [daemon's config file](#)

Bash

```
sudo nano /etc/docker/daemon.json
```

2. Set the default logging driver to the `local` logging driver as shown in the example.

JSON

```
{
  "log-driver": "local"
}
```

3. Restart the container engine for the changes to take effect.

Bash

```
sudo systemctl restart docker
```

Install the IoT Edge runtime

The IoT Edge service provides and maintains security standards on the IoT Edge device. The service starts on every boot and bootstraps the device by starting the rest of the IoT Edge runtime.

ⓘ Note

Beginning with version 1.2, the [IoT identity service](#) handles identity provisioning and management for IoT Edge and for other device components that need to communicate with IoT Hub.

The steps in this section represent the typical process to install the latest IoT Edge version on a device that has internet connection. If you need to install a specific version, like a pre-release version, or need to install while offline, follow the [Offline or specific version installation](#) steps later in this article.

💡 Tip

If you already have an IoT Edge device running an older version and want to upgrade to the latest release, use the steps in [Update the IoT Edge security daemon and runtime](#). Later versions are sufficiently different from previous versions of IoT Edge that specific steps are necessary to upgrade.

Ubuntu

Install the latest version of IoT Edge and the IoT identity service package (if you're not already [up-to-date](#)):

- 22.04:

Bash

```
sudo apt-get update; \
    sudo apt-get install aziot-edge
```

- 20.04:

Bash

```
sudo apt-get update; \
```

```
sudo apt-get install aziot-edge defender-iot-micro-agent-edge
```

The optional `defender-iot-micro-agent-edge` package includes the Microsoft Defender for IoT security micro-agent that provides endpoint visibility into security posture management, vulnerabilities, threat detection, fleet management and more to help you secure your IoT Edge devices. It's recommended to install the micro agent with the Edge agent to enable security monitoring and hardening of your Edge devices. To learn more about Microsoft Defender for IoT, see [What is Microsoft Defender for IoT for device builders](#).

Provision the device with its cloud identity

Now that the container engine and the IoT Edge runtime are installed on your device, you're ready to set up the device with its cloud identity and authentication information.

Ubuntu / Debian / RHEL

You can configure your IoT Edge device with symmetric key authentication using the following command:

Bash

```
sudo iotedge config mp --connection-string  
'PASTE_DEVICE_CONNECTION_STRING_HERE'
```

This `iotedge config mp` command creates a configuration file on the device and enters your connection string in the configuration file.

1. Apply the configuration changes.

Bash

```
sudo iotedge config apply
```

2. To view the configuration file, you can open it:

Bash

```
sudo nano /etc/aziot/config.toml
```

Deploy modules

To deploy your IoT Edge modules, go to your IoT hub in the Azure portal, then:

1. Select **Devices** from the IoT Hub menu.
2. Select your device to open its page.
3. Select the **Set Modules** tab.
4. Since we want to deploy the IoT Edge default modules (`edgeAgent` and `edgeHub`), we don't need to add any modules to this pane, so select **Review + create** at the bottom.
5. You see the JSON confirmation of your modules. Select **Create** to deploy the modules.

For more information, see [Deploy a module](#).

Verify successful configuration

Verify that the runtime was successfully installed and configured on your IoT Edge device.

Tip

You need elevated privileges to run `iotedge` commands. Once you sign out of your machine and sign back in the first time after installing the IoT Edge runtime, your permissions are automatically updated. Until then, use `sudo` in front of the commands.

1. Check to see that the IoT Edge system service is running.

```
Bash
```

```
sudo iotedge system status
```

A successful status response shows the `aziot` services as running or ready.

2. If you need to troubleshoot the service, retrieve the service logs.

```
Bash
```

```
sudo iotedge system logs
```

3. Use the `check` tool to verify configuration and connection status of the device.

Bash

```
sudo iotedge check
```

You can expect a range of responses that may include **OK** (green), **Warning** (yellow), or **Error** (red). For troubleshooting common errors, see [Solutions to common issues for Azure IoT Edge](#).

```
Configuration checks
-----
✓ aziot-edged configuration is well-formed - OK
✓ configuration up-to-date with config.toml - OK
✓ container engine is installed and functional - OK
✓ configuration has correct URIs for daemon mgmt endpoint - OK
✓ aziot-edge package is up-to-date - OK
✓ container time is close to host time - OK
» DNS server - Warning
  Container engine is not configured with DNS server setting, which may impact connectivity to IoT Hub.
  Please see https://aka.ms/iotedge-prod-checklist-dns for best practices.
  You can ignore this warning if you are setting DNS server per module in the Edge deployment.
!! production readiness: logs policy - Warning
  Container engine is not configured to rotate module logs which may cause it run out of disk space.
  Please see https://aka.ms/iotedge-prod-checklist-logs for best practices.
  You can ignore this warning if you are setting log policy per module in the Edge deployment.
✗ production readiness: Edge Agent's storage directory is persisted on the host filesystem - Error
  Could not check current state of edgeAgent container
✗ production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error
  Could not check current state of edgeHub container
✓ Agent image is valid and can be pulled from upstream - OK
✓ proxy settings are consistent in aziot-edged, aziot-identityd, moby daemon and config.toml - OK
```

💡 Tip

Always use `sudo` to run the check tool, even after your permissions are updated. The tool needs elevated privileges to access the config file to verify configuration status.

❗ Note

On a newly provisioned device, you may see an error related to IoT Edge Hub:

✗ **production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error** Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module is not yet running. Be sure your IoT Edge modules were deployed in the previous steps. Deployment resolves this error.

Alternatively, you may see a status code as 417 -- The device's deployment configuration is not set. Once your modules are deployed, this status will change.

- When the service starts for the first time, you should only see the `edgeAgent` module running. The `edgeAgent` module runs by default and helps to install and start any additional modules that you deploy to your device.

Check that your device and modules are deployed and running, by viewing your device page in the Azure portal.

Modules					
Name	Type	Specified in Deployment	Reported	Running	
\$edgeAgent	IoT Edge System Module	✓ Yes	✓ Yes	running	
\$edgeHub	IoT Edge System Module	✓ Yes	✓ Yes	running	

Once your modules are deployed and running, list them in your device or virtual machine with the following command:

```
Bash  
sudo iotedge list
```

Offline or specific version installation (optional)

The steps in this section are for scenarios not covered by the standard installation steps. This may include:

- Installing IoT Edge while offline
- Installing a release candidate version

Use the steps in this section if you want to install a [specific version of the Azure IoT Edge runtime](#) that isn't available through your package manager. The Microsoft package list only contains a limited set of recent versions and their sub-versions, so these steps are for anyone who wants to install an older version or a release candidate version.

If you are using Ubuntu snaps, you can download a snap and install it offline. For more information, see [Download snaps and install offline ↗](#).

Using curl commands, you can target the component files directly from the IoT Edge GitHub repository.

1. Navigate to the [Azure IoT Edge releases](#), and find the release version that you want to target.
2. Expand the **Assets** section for that version.
3. Every release should have new files for IoT Edge and the identity service. If you're going to install IoT Edge on an offline device, download these files ahead of time. Otherwise, use the following commands to update those components.
 - a. Find the **aziot-identity-service** file that matches your IoT Edge device's architecture. Right-click on the file link and copy the link address.
 - b. Use the copied link in the following command to install that version of the identity service:

Ubuntu / Debian

Bash

```
curl -L <identity service link> -o aziot-identity-service.deb &&
sudo apt-get install ./aziot-identity-service.deb
```

- c. Find the **aziot-edge** file that matches your IoT Edge device's architecture. Right-click on the file link and copy the link address.
- d. Use the copied link in the following command to install that version of IoT Edge.

Ubuntu / Debian

Bash

```
curl -L <iotedge link> -o aziot-edge.deb && sudo apt-get install
./aziot-edge.deb
```

Uninstall IoT Edge

If you want to remove the IoT Edge installation from your device, use the following commands.

Remove the IoT Edge runtime.

Ubuntu / Debian

Bash

```
sudo apt-get autoremove --purge aziot-edge
```

Leave out the `--purge` flag if you plan to reinstall IoT Edge and use the same configuration information in the future. The `--purge` flag deletes all the files associated with IoT Edge, including your configuration files.

When the IoT Edge runtime is removed, any containers that it created are stopped but still exist on your device. View all containers to see which ones remain.

Bash

```
sudo docker ps -a
```

Delete the containers from your device, including the two runtime containers.

Bash

```
sudo docker rm -f <container ID>
```

Finally, remove the container runtime from your device.

Ubuntu / Debian

Bash

```
sudo apt-get autoremove --purge moby-engine
```

Next steps

Continue to [deploy IoT Edge modules](#) to learn how to deploy modules onto your device.

Create and provision IoT Edge devices at scale on Linux using X.509 certificates

Article • 06/13/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides end-to-end instructions for autoprovisioning one or more Linux IoT Edge devices using X.509 certificates. You can automatically provision Azure IoT Edge devices with the [Azure IoT Hub device provisioning service \(DPS\)](#). If you're unfamiliar with the process of autoprovisioning, review the [provisioning overview](#) before continuing.

The tasks are as follows:

1. Generate certificates and keys.
2. Create either an **individual enrollment** for a single device or a **group enrollment** for a set of devices.
3. Install the IoT Edge runtime and register the device with IoT Hub.

Using X.509 certificates as an attestation mechanism is an excellent way to scale production and simplify device provisioning. Typically, X.509 certificates are arranged in a certificate chain of trust. Starting with a self-signed or trusted root certificate, each certificate in the chain signs the next lower certificate. This pattern creates a delegated chain of trust from the root certificate down through each intermediate certificate to the final downstream device certificate installed on a device.

Tip

If your device has a Hardware Security Module (HSM) such as a TPM 2.0, then we recommend storing the X.509 keys securely in the HSM. Learn more about how to implement the zero-touch provisioning at scale described in [this blueprint](#)  with the [iotedge-tpm2cloud](#)  sample.

Prerequisites

Cloud resources

- An active IoT hub
- An instance of the IoT Hub device provisioning service in Azure, linked to your IoT hub
 - If you don't have a device provisioning service instance, you can follow the instructions in the [Create a new IoT Hub device provisioning service](#) and [Link the IoT hub and your device provisioning service](#) sections of the IoT Hub device provisioning service quickstart.
 - After you have the device provisioning service running, copy the value of **ID Scope** from the overview page. You use this value when you configure the IoT Edge runtime.

Device requirements

A physical or virtual Linux device to be the IoT Edge device.

Generate device identity certificates

The device identity certificate is a downstream device certificate that connects through a certificate chain of trust to the top X.509 certificate authority (CA) certificate. The device identity certificate must have its common name (CN) set to the device ID that you want the device to have in your IoT hub.

Device identity certificates are only used for provisioning the IoT Edge device and authenticating the device with Azure IoT Hub. They aren't signing certificates, unlike the CA certificates that the IoT Edge device presents to modules or downstream devices for verification. For more information, see [Azure IoT Edge certificate usage detail](#).

After you create the device identity certificate, you should have two files: a .cer or .pem file that contains the public portion of the certificate, and a .cer or .pem file with the private key of the certificate. If you plan to use group enrollment in DPS, you also need the public portion of an intermediate or root CA certificate in the same certificate chain of trust.

You need the following files to set up automatic provisioning with X.509:

- The device identity certificate and its private key certificate. The device identity certificate is uploaded to DPS if you create an individual enrollment. The private

key is passed to the IoT Edge runtime.

- A full chain certificate, which should have at least the device identity and the intermediate certificates in it. The full chain certificate is passed to the IoT Edge runtime.
- An intermediate or root CA certificate from the certificate chain of trust. This certificate is uploaded to DPS if you create a group enrollment.

Use test certificates (optional)

If you don't have a certificate authority available to create new identity certs and want to try out this scenario, the Azure IoT Edge git repository contains scripts that you can use to generate test certificates. These certificates are designed for development testing only, and must not be used in production.

To create test certificates, follow the steps in [Create demo certificates to test IoT Edge device features](#). Complete the two required sections to set up the certificate generation scripts and to create a root CA certificate. Then, follow the steps to create a device identity certificate. When you're finished, you should have the following certificate chain and key pair:

- `<WRKDIR>/certs/iot-edge-device-identity-<name>-full-chain.cert.pem`
- `<WRKDIR>/private/iot-edge-device-identity-<name>.key.pem`

You need both these certificates on the IoT Edge device. If you're going to use individual enrollment in DPS, then you upload the .cert.pem file. If you're going to use group enrollment in DPS, then you also need an intermediate or root CA certificate in the same certificate chain of trust to upload. If you're using demo certs, use the `<WRKDIR>/certs/azure-iot-test-only.root.ca.cert.pem` certificate for group enrollment.

Create a DPS enrollment

Use your generated certificates and keys to create an enrollment in DPS for one or more IoT Edge devices.

If you are looking to provision a single IoT Edge device, create an [individual enrollment](#). If you need multiple devices provisioned, follow the steps for creating a [DPS group enrollment](#).

When you create an enrollment in DPS, you have the opportunity to declare an [Initial Device Twin State](#). In the device twin, you can set tags to group devices by any metric you need in your solution, like region, environment, location, or device type. These tags are used to create [automatic deployments](#).

For more information about enrollments in the device provisioning service, see [How to manage device enrollments](#).

Individual enrollment

Create a DPS individual enrollment

Individual enrollments take the public portion of a device's identity certificate and match that to the certificate on the device.

Tip

The steps in this article are for the Azure portal, but you can also create individual enrollments using the Azure CLI. For more information, see [az iot dps enrollment](#). As part of the CLI command, use the **edge-enabled** flag to specify that the enrollment is for an IoT Edge device.

1. In the [Azure portal](#), navigate to your instance of IoT Hub device provisioning service.
2. Under **Settings**, select **Manage enrollments**.
3. Select **Add individual enrollment** then complete the following steps to configure the enrollment:
 - **Mechanism:** Select **X.509**.
 - **Primary Certificate .pem or .cer file:** Upload the public file from the device identity certificate. If you used the scripts to generate a test certificate, choose the following file:
`<WRKDIR>\certs\iot-edge-device-identity-<name>.cert.pem`
 - **IoT Hub Device ID:** Provide an ID for your device if you'd like. You can use device IDs to target an individual device for module deployment. If you don't provide a device ID, the common name (CN) in the X.509 certificate is used.
 - **IoT Edge device:** Select **True** to declare that the enrollment is for an IoT Edge device.

- **Select the IoT hubs this device can be assigned to:** Choose the linked IoT hub that you want to connect your device to. You can choose multiple hubs, and the device will be assigned to one of them according to the selected allocation policy.
- **Initial Device Twin State:** Add a tag value to be added to the device twin if you'd like. You can use tags to target groups of devices for automatic deployment. For example:

JSON

```
{  
  "tags": {  
    "environment": "test"  
  },  
  "properties": {  
    "desired": {}  
  }  
}
```

4. Select Save.

Under **Manage Enrollments**, you can see the **Registration ID** for the enrollment you just created. Make note of it, as it can be used when you provision your device.

Now that an enrollment exists for this device, the IoT Edge runtime can automatically provision the device during installation.

Install IoT Edge

In this section, you prepare your Linux virtual machine or physical device for IoT Edge. Then, you install IoT Edge.

Run the following commands to add the package repository and then add the Microsoft package signing key to your list of trusted keys.

ⓘ Important

On June 30, 2022 Raspberry Pi OS Stretch was retired from the Tier 1 OS support list. To avoid potential security vulnerabilities update your host OS to Bullseye.

Installing can be done with a few commands. Open a terminal and run the following commands:

- 22.04:

```
Bash
```

```
wget https://packages.microsoft.com/config/ubuntu/22.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

- 20.04:

```
Bash
```

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

For more information about operating system versions, see [Azure IoT Edge supported platforms](#).

 **Note**

Azure IoT Edge software packages are subject to the license terms located in each package (`usr/share/doc/{package-name}` or the `LICENSE` directory). Read the license terms prior to using a package. Your installation and use of a package constitutes your acceptance of these terms. If you don't agree with the license terms, don't use that package.

Install a container engine

Azure IoT Edge relies on an [OCI](#)-compatible container runtime. For production scenarios, we recommend that you use the Moby engine. The Moby engine is the container engine officially supported with IoT Edge. Docker CE/EE container images are compatible with the Moby runtime. If you are using Ubuntu Core snaps, the Docker snap is serviced by Canonical and supported for production scenarios.

Ubuntu

Install the Moby engine.

Bash

```
sudo apt-get update; \
  sudo apt-get install moby-engine
```

By default, the container engine doesn't set container log size limits. Over time, this can lead to the device filling up with logs and running out of disk space. However, you can configure your log to show locally, though it's optional. To learn more about logging configuration, see [Production Deployment Checklist](#).

The following steps show you how to configure your container to use [local logging driver](#) as the logging mechanism.

Ubuntu / Debian / RHEL

1. Create or edit the existing Docker [daemon's config file](#)

Bash

```
sudo nano /etc/docker/daemon.json
```

2. Set the default logging driver to the `local` logging driver as shown in the example.

JSON

```
{
  "log-driver": "local"
}
```

3. Restart the container engine for the changes to take effect.

Bash

```
sudo systemctl restart docker
```

Install the IoT Edge runtime

The IoT Edge service provides and maintains security standards on the IoT Edge device. The service starts on every boot and bootstraps the device by starting the rest of the IoT Edge runtime.

ⓘ Note

Beginning with version 1.2, the [IoT identity service](#) handles identity provisioning and management for IoT Edge and for other device components that need to communicate with IoT Hub.

The steps in this section represent the typical process to install the latest IoT Edge version on a device that has internet connection. If you need to install a specific version, like a pre-release version, or need to install while offline, follow the [Offline or specific version installation](#) steps later in this article.

💡 Tip

If you already have an IoT Edge device running an older version and want to upgrade to the latest release, use the steps in [Update the IoT Edge security daemon and runtime](#). Later versions are sufficiently different from previous versions of IoT Edge that specific steps are necessary to upgrade.

Ubuntu

Install the latest version of IoT Edge and the IoT identity service package (if you're not already [up-to-date](#)):

- 22.04:

Bash

```
sudo apt-get update; \
    sudo apt-get install aziot-edge
```

- 20.04:

Bash

```
sudo apt-get update; \
```

```
sudo apt-get install aziot-edge defender-iot-micro-agent-edge
```

The optional `defender-iot-micro-agent-edge` package includes the Microsoft Defender for IoT security micro-agent that provides endpoint visibility into security posture management, vulnerabilities, threat detection, fleet management and more to help you secure your IoT Edge devices. It's recommended to install the micro agent with the Edge agent to enable security monitoring and hardening of your Edge devices. To learn more about Microsoft Defender for IoT, see [What is Microsoft Defender for IoT for device builders](#).

Provision the device with its cloud identity

Once the runtime is installed on your device, configure the device with the information it uses to connect to the device provisioning service and IoT Hub.

Have the following information ready:

- The DPS ID Scope value. You can retrieve this value from the overview page of your DPS instance in the Azure portal.
- The device identity certificate chain file on the device.
- The device identity key file on the device.

Create a configuration file for your device based on a template file that is provided as part of the IoT Edge installation.

Ubuntu / Debian / RHEL

Bash

```
sudo cp /etc/aziot/config.toml.edge.template /etc/aziot/config.toml
```

Open the configuration file on the IoT Edge device.

Bash

```
sudo nano /etc/aziot/config.toml
```

Find the **Provisioning** section of the file. Uncomment the lines for DPS provisioning with X.509 certificate, and make sure any other provisioning lines are commented out.

```
toml
```

```
# DPS provisioning with X.509 certificate
[provisioning]
source = "dps"
global_endpoint = "https://global.azure-devices-provisioning.net"
id_scope = "SCOPE_ID_HERE"

# Uncomment to send a custom payload during DPS registration
# payload = { uri = "PATH_TO_JSON_FILE" }

[provisioning.attestation]
method = "x509"
registration_id = "REGISTRATION_ID_HERE"

identity_cert = "DEVICE_IDENTITY_CERTIFICATE_HERE" # For example,
"file:///var/aziot/device-id.pem"
identity_pk = "DEVICE_IDENTITY_PRIVATE_KEY_HERE"    # For example,
"file:///var/aziot/device-id.key"

# auto_reprovisioning_mode = Dynamic
```

1. Update the value of `id_scope` with the scope ID you copied from your instance of DPS.
2. Provide a `registration_id` for the device, which is the ID that the device has in IoT Hub. The registration ID must match the common name (CN) of the identity certificate.
3. Update the values of `identity_cert` and `identity_pk` with your certificate and key information.

The identity certificate value can be provided as a file URI, or can be dynamically issued using EST or a local certificate authority. Uncomment only one line, based on the format you choose to use.

The identity private key value can be provided as a file URI or a PKCS#11 URI. Uncomment only one line, based on the format you choose to use.

If you use any PKCS#11 URIs, find the **PKCS#11** section in the config file and provide information about your PKCS#11 configuration.

For more information about certificates, see [Manage IoT Edge certificates](#).

For more information about provisioning configuration settings, see [Configure IoT Edge device settings](#).

4. Optionally, find the auto reprovisioning mode section of the file. Use the `auto_reprovisioning_mode` parameter to configure your device's reprovisioning behavior. **Dynamic** - Reprovision when the device detects that it may have been moved from one IoT Hub to another. This is the default. **AlwaysOnStartup** - Reprovision when the device is rebooted or a crash causes the daemons to restart. **OnErrorOnly** - Never trigger device reprovisioning automatically. Each mode has an implicit device reprovisioning fallback if the device is unable to connect to IoT Hub during identity provisioning due to connectivity errors. For more information, see [IoT Hub device reprovisioning concepts](#).

5. Optionally, uncomment the `payload` parameter to specify the path to a local JSON file. The contents of the file is [sent to DPS as additional data](#) when the device registers. This is useful for [custom allocation](#). For example, if you want to allocate your devices based on an IoT Plug and Play model ID without human intervention.

6. Save and close the file.

Apply the configuration changes that you made to IoT Edge.

```
Ubuntu / Debian / RHEL

Bash

sudo iotedge config apply
```

Verify successful installation

If the runtime started successfully, you can go into your IoT Hub and start deploying IoT Edge modules to your device.

Individual enrollment

You can verify that the individual enrollment that you created in device provisioning service was used. Navigate to your device provisioning service instance in the Azure portal. Open the enrollment details for the individual enrollment that you created. Notice that the status of the enrollment is **assigned** and the device ID is listed.

Use the following commands on your device to verify that the IoT Edge installed and started successfully.

Check the status of the IoT Edge service.

```
cmd/sh
```

```
sudo iotedge system status
```

Examine service logs.

```
cmd/sh
```

```
sudo iotedge system logs
```

List running modules.

```
cmd/sh
```

```
sudo iotedge list
```

Next steps

The device provisioning service enrollment process lets you set the device ID and device twin tags at the same time as you provision the new device. You can use those values to target individual devices or groups of devices using automatic device management.

Learn how to [Deploy and monitor IoT Edge modules at scale using the Azure portal](#) or [using Azure CLI](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Create and provision IoT Edge devices at scale with a TPM on Linux

Article • 04/17/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides instructions for autoprovisioning an Azure IoT Edge for Linux device by using a Trusted Platform Module (TPM). You can automatically provision IoT Edge devices with the [Azure IoT Hub device provisioning service](#). If you're unfamiliar with the process of autoprovisioning, review the [provisioning overview](#) before you continue.

This article outlines two methodologies. Select your preference based on the architecture of your solution:

- Autoprovision a Linux device with physical TPM hardware.
- Autoprovision a Linux virtual machine (VM) with a simulated TPM running on a Windows development machine with Hyper-V enabled. We recommend using this methodology only as a testing scenario. A simulated TPM doesn't offer the same security as a physical TPM.

Instructions differ based on your methodology, so make sure you're on the correct tab going forward.

Physical device

The tasks are as follows:

1. Retrieve provisioning information for your TPM.
2. Create an individual enrollment for your device in an instance of the IoT Hub device provisioning service.
3. Install the IoT Edge runtime and connect the device to the IoT hub.

Prerequisites

Cloud resources

- An active IoT hub
- An instance of the IoT Hub device provisioning service in Azure, linked to your IoT hub
 - If you don't have a device provisioning service instance, you can follow the instructions in the [Create a new IoT Hub device provisioning service](#) and [Link the IoT hub and your device provisioning service](#) sections of the IoT Hub device provisioning service quickstart.
 - After you have the device provisioning service running, copy the value of **ID Scope** from the overview page. You use this value when you configure the IoT Edge runtime.

Device requirements

Physical device

A physical Linux device to be the IoT Edge device.

If you are a device manufacturer, then refer to guidance on [integrating a TPM into the manufacturing process](#).

ⓘ Note

TPM 2.0 is required when you use TPM attestation with the device provisioning service.

You can only create individual, not group, device provisioning service enrollments when you use a TPM.

Set up your device

Physical device

If you're using a physical Linux device with a TPM, there are no extra steps to set up your device.

You're ready to continue.

Retrieve provisioning information for your TPM

ⓘ Note

This article previously used the `tpm_device_provision` tool from the IoT C SDK to generate provisioning info. If you relied on that tool previously, then be aware the steps below generate a different registration ID for the same public endorsement key. If you need to recreate the registration ID as before then refer to how the C SDK's [tpm_device_provision tool](#) generates it. Be sure the registration ID for the individual enrollment in DPS matches the registration ID the IoT Edge device is configured to use.

In this section, you use the TPM2 software tools to retrieve the endorsement key for your TPM and then generate a unique registration ID. This section corresponds with [Step 3: Device has firmware and software installed](#) in the process for [integrating a TPM into the manufacturing process](#).

Install the TPM2 Tools

Sign in to your device, and install the `tpm2-tools` package.

Ubuntu / Debian

Bash

```
sudo apt-get install tpm2-tools
```

Run the following script to read the endorsement key, creating one if it does not already exist.

Bash

```
#!/bin/sh
if [ "$USER" != "root" ]; then
    SUDO="sudo "
fi

$SUDO tpm2_readpublic -Q -c 0x81010001 -o ek.pub 2> /dev/null
if [ $? -gt 0 ]; then
```

```

# Create the endorsement key (EK)
$SUDO tpm2_createek -c 0x81010001 -G rsa -u ek.pub

# Create the storage root key (SRK)
$SUDO tpm2_createprimary -Q -C o -c srk.ctx > /dev/null

# make the SRK persistent
$SUDO tpm2_evictcontrol -c srk.ctx 0x81000001 > /dev/null
fi

printf "Gathering the registration information...\n\nRegistration
Id:\n%s\n\nEndorsement Key:\n%s\n" $(sha256sum -b ek.pub | cut -d' ' -f1 |
sed -e 's/[[:alnum:]]//g') $(base64 -w0 ek.pub)
$SUDO rm ek.pub srk.ctx 2> /dev/null

```

The output window displays the device's **Endorsement key** and a unique **Registration ID**. Copy these values for use later when you create an individual enrollment for your device in the device provisioning service.

After you have your registration ID and endorsement key, you're ready to continue.

💡 Tip

If you don't want to use the TPM2 software tools to retrieve the information, you need to find another way to obtain the provisioning information. The endorsement key, which is unique to each TPM chip, is obtained from the TPM chip manufacturer associated with it. You can derive a unique registration ID for your TPM device. For example, as shown above you can create an SHA-256 hash of the endorsement key.

Create a device provisioning service enrollment

Use your TPM's provisioning information to create an individual enrollment in the device provisioning service.

When you create an enrollment in the device provisioning service, you have the opportunity to declare an **Initial Device Twin State**. In the device twin, you can set tags to group devices by any metric used in your solution, like region, environment, location, or device type. These tags are used to create [automatic deployments](#).

💡 Tip

The steps in this article are for the Azure portal, but you can also create individual enrollments by using the Azure CLI. For more information, see [az iot dps enrollment](#). As part of the CLI command, use the `edge-enabled` flag to specify that the enrollment is for an IoT Edge device.

1. In the [Azure portal](#), go to your instance of the IoT Hub device provisioning service.
2. Under **Settings**, select **Manage enrollments**.
3. Select **Add individual enrollment**, and then complete the following steps to configure the enrollment:
 - a. For **Mechanism**, select **TPM**.
 - b. Provide the **Endorsement key** and **Registration ID** that you copied from your VM or physical device.
 - c. Provide an ID for your device if you want. If you don't provide a device ID, the registration ID is used.
 - d. Select **True** to declare that your VM or physical device is an IoT Edge device.
 - e. Choose the linked IoT hub that you want to connect your device to, or select **Link to new IoT Hub**. You can choose multiple hubs, and the device will be assigned to one of them according to the selected assignment policy.
 - f. Add a tag value to the **Initial Device Twin State** if you want. You can use tags to target groups of devices for module deployment. For more information, see [Deploy IoT Edge modules at scale](#).
 - g. Select **Save**.

Now that an enrollment exists for this device, the IoT Edge runtime can automatically provision the device during installation.

Install IoT Edge

In this section, you prepare your Linux virtual machine or physical device for IoT Edge. Then, you install IoT Edge.

Run the following commands to add the package repository and then add the Microsoft package signing key to your list of trusted keys.

ⓘ Important

On June 30, 2022 Raspberry Pi OS Stretch was retired from the Tier 1 OS support list. To avoid potential security vulnerabilities update your host OS to Bullseye.

For [tier 2 supported platform operating systems](#), installation packages are made available at [Azure IoT Edge releases](#). See the installation steps in [Offline or specific version installation](#).

Ubuntu

Installing can be done with a few commands. Open a terminal and run the following commands:

- 24.04:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/24.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

- 22.04:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/22.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

- 20.04:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb
sudo dpkg -i packages-microsoft-prod.deb
rm packages-microsoft-prod.deb
```

For more information about operating system versions, see [Azure IoT Edge supported platforms](#).

Note

Azure IoT Edge software packages are subject to the license terms located in each package (`usr/share/doc/{package-name}` or the `LICENSE` directory). Read the license terms prior to using a package. Your installation and use of a package constitutes your acceptance of these terms. If you don't agree with the license terms, don't use that package.

Install a container engine

Azure IoT Edge relies on an [OCI](#)-compatible container runtime. For production scenarios, we recommend that you use the Moby engine. The Moby engine is the container engine officially supported with IoT Edge. Docker CE/EE container images are compatible with the Moby runtime. If you are using Ubuntu Core snaps, the Docker snap is serviced by Canonical and supported for production scenarios.

Ubuntu

Install the Moby engine.

Bash

```
sudo apt-get update; \
  sudo apt-get install moby-engine
```

By default, the container engine doesn't set container log size limits. Over time, this can lead to the device filling up with logs and running out of disk space. However, you can configure your log to show locally, though it's optional. To learn more about logging configuration, see [Production Deployment Checklist](#).

The following steps show you how to configure your container to use [local logging driver](#) as the logging mechanism.

Ubuntu / Debian / RHEL

1. Create or edit the existing Docker [daemon's config file](#)

Bash

```
sudo nano /etc/docker/daemon.json
```

- Set the default logging driver to the `local` logging driver as shown in the example.

JSON

```
{  
  "log-driver": "local"  
}
```

- Restart the container engine for the changes to take effect.

Bash

```
sudo systemctl restart docker
```

Install the IoT Edge runtime

The IoT Edge service provides and maintains security standards on the IoT Edge device. The service starts on every boot and bootstraps the device by starting the rest of the IoT Edge runtime.

Note

Beginning with version 1.2, the [IoT identity service](#) handles identity provisioning and management for IoT Edge and for other device components that need to communicate with IoT Hub.

The steps in this section represent the typical process to install the latest IoT Edge version on a device that has internet connection. If you need to install a specific version, like a pre-release version, or need to install while offline, follow the [Offline or specific version installation](#) steps later in this article.

Tip

If you already have an IoT Edge device running an older version and want to upgrade to the latest release, use the steps in [Update the IoT Edge security](#)

daemon and runtime. Later versions are sufficiently different from previous versions of IoT Edge that specific steps are necessary to upgrade.

Ubuntu

Install the latest version of IoT Edge and the IoT identity service package (if you're not already [up-to-date](#)):

- 22.04:

Bash

```
sudo apt-get update; \
    sudo apt-get install aziot-edge
```

- 20.04:

Bash

```
sudo apt-get update; \
    sudo apt-get install aziot-edge defender-iot-micro-agent-edge
```

The optional `defender-iot-micro-agent-edge` package includes the Microsoft Defender for IoT security micro-agent that provides endpoint visibility into security posture management, vulnerabilities, threat detection, fleet management and more to help you secure your IoT Edge devices. It's recommended to install the micro agent with the Edge agent to enable security monitoring and hardening of your Edge devices. To learn more about Microsoft Defender for IoT, see [What is Microsoft Defender for IoT for device builders](#).

Provision the device with its cloud identity

After the runtime is installed on your device, configure the device with the information it uses to connect to the device provisioning service and IoT Hub.

Know your device provisioning service **ID Scope** and device **Registration ID** that were gathered previously.

Create a configuration file for your device based on a template file that's provided as part of the IoT Edge installation.

Ubuntu / Debian / RHEL

Bash

```
sudo cp /etc/aziot/config.toml.edge.template /etc/aziot/config.toml
```

Open the configuration file on the IoT Edge device.

Bash

```
sudo nano /etc/aziot/config.toml
```

1. Find the provisioning configurations section of the file. Uncomment the lines for TPM provisioning, and make sure any other provisioning lines are commented out.

```
toml

# DPS provisioning with TPM
[provisioning]
source = "dps"
global_endpoint = "https://global.azure-devices-provisioning.net"
id_scope = "DPS_ID_SCOPE_HERE"

# Uncomment to send a custom payload during DPS registration
# payload = { uri = "PATH_TO_JSON_FILE" }

[provisioning.attestation]
method = "tpm"
registration_id = "REGISTRATION_ID_HERE"

# auto_reprovisioning_mode = Dynamic
```

2. Update the values of `id_scope` and `registration_id` with your device provisioning service and device information. The `scope_id` value is the **ID Scope** from your device provisioning service instance's overview page.

For more information about provisioning configuration settings, see [Configure IoT Edge device settings](#).

3. Optionally, find the auto reprovisioning mode section of the file. Use the `auto_reprovisioning_mode` parameter to configure your device's reprovisioning behavior. **Dynamic** - Reprovision when the device detects that it may have been moved from one IoT Hub to another. This is the default. **AlwaysOnStartup** - Reprovision when the device is rebooted or a crash causes the daemons to restart.

OnErrorOnly - Never trigger device reprovisioning automatically. Each mode has an implicit device reprovisioning fallback if the device is unable to connect to IoT Hub during identity provisioning due to connectivity errors. For more information, see [IoT Hub device reprovisioning concepts](#).

4. Optionally, uncomment the `payload` parameter to specify the path to a local JSON file. The contents of the file is sent to DPS as additional data when the device registers. This is useful for [custom allocation](#). For example, if you want to allocate your devices based on an IoT Plug and Play model ID without human intervention.
5. Save and close the file.

Give IoT Edge access to the TPM

The IoT Edge runtime relies on a TPM service that brokers access to a device's TPM. This service needs to access the TPM to automatically provision your device.

You can give access to the TPM by overriding the systemd settings so that the `aziottpm` service has root privileges. If you don't want to elevate the service privileges, you can also use the following steps to manually provide TPM access.

1. Create a new rule that gives the IoT Edge runtime access to `tpm0` and `tpmrm0`.

```
Bash
```

```
sudo touch /etc/udev/rules.d/tpmaccess.rules
```

2. Open the rules file.

```
Bash
```

```
sudo nano /etc/udev/rules.d/tpmaccess.rules
```

3. Copy the following access information into the rules file. The `tpmrm0` might not be present on devices that use a kernel earlier than 4.12. Devices that don't have `tpmrm0` will safely ignore that rule.

```
input
```

```
# allow aziottpm access to tpm0 and tpmrm0
KERNEL=="tpm0", SUBSYSTEM=="tpm", OWNER="aziottpm", MODE="0660"
KERNEL=="tpmrm0", SUBSYSTEM=="tpmrm", OWNER="aziottpm", MODE="0660"
```

4. Save and exit the file.

5. Trigger the `udev` system to evaluate the new rule.

Bash

```
/bin/udevadm trigger --subsystem-match=tpm --subsystem-match=tpmrm
```

6. Verify that the rule was successfully applied.

Bash

```
ls -l /dev/tpm*
```

Successful output appears as follows:

Output

```
crw-rw---- 1 root aziottpm 10, 224 Jul 20 16:27 /dev/tpm0
crw-rw---- 1 root aziottpm 10, 224 Jul 20 16:27 /dev/tpmrm0
```

If you don't see that the correct permissions applied, try rebooting your machine to refresh `udev`.

7. Apply the configuration changes that you made on the device.

Ubuntu / Debian / RHEL

Bash

```
sudo iotedge config apply
```

Verify successful installation

If you didn't already, apply the configuration changes that you made on the device.

Bash

```
sudo iotedge config apply
```

Check to see that the IoT Edge runtime is running.

```
Bash
```

```
sudo iotedge system status
```

Examine daemon logs.

```
cmd/sh
```

```
sudo iotedge system logs
```

If you see provisioning errors, it might be that the configuration changes haven't taken effect yet. Try restarting the IoT Edge daemon.

```
Bash
```

```
sudo systemctl daemon-reload
```

Or, try restarting your VM to see if the changes take effect on a fresh start.

If the runtime started successfully, you can go into your IoT hub and see that your new device was automatically provisioned. Now your device is ready to run IoT Edge modules.

List running modules.

```
cmd/sh
```

```
iotedge list
```

You can verify that the individual enrollment that you created in the device provisioning service was used. Go to your device provisioning service instance in the Azure portal. Open the enrollment details for the individual enrollment that you created. Notice that the status of the enrollment is **assigned** and the device ID is listed.

Next steps

The device provisioning service enrollment process lets you set the device ID and device twin tags at the same time as you provision the new device. You can use those values to target individual devices or groups of devices by using automatic device management.

Learn how to [deploy and monitor IoT Edge modules at scale](#) by using the [Azure portal](#) or [the Azure CLI](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Create and provision IoT Edge devices at scale on Linux using symmetric key

Article • 03/27/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides end-to-end instructions for autoprovisioning one or more Linux IoT Edge devices using symmetric keys. You can automatically provision Azure IoT Edge devices with the [Azure IoT Hub device provisioning service \(DPS\)](#). If you're unfamiliar with the process of autoprovisioning, review the [provisioning overview](#) before continuing.

The tasks are as follows:

1. Create either an [individual enrollment](#) for a single device or a [group enrollment](#) for a set of devices.
2. Install the IoT Edge runtime and connect to the IoT Hub.

Tip

For a simplified experience, try the [Azure IoT Edge configuration tool](#). This command-line tool, currently in public preview, installs IoT Edge on your device and provisions it using DPS and symmetric key attestation.

Symmetric key attestation is a simple approach to authenticating a device with a device provisioning service instance. This attestation method represents a "Hello world" experience for developers who are new to device provisioning, or do not have strict security requirements. Device attestation using a [TPM](#) or [X.509 certificates](#) is more secure, and should be used for more stringent security requirements.

Prerequisites

Cloud resources

- An active IoT hub
- An instance of the IoT Hub device provisioning service in Azure, linked to your IoT hub
 - If you don't have a device provisioning service instance, you can follow the instructions in the [Create a new IoT Hub device provisioning service](#) and [Link the IoT hub and your device provisioning service](#) sections of the IoT Hub device provisioning service quickstart.
 - After you have the device provisioning service running, copy the value of **ID Scope** from the overview page. You use this value when you configure the IoT Edge runtime.

Device requirements

A physical or virtual Linux device to be the IoT Edge device.

You will need to define a *unique registration ID* to identify each device. You can use the MAC address, serial number, or any unique information from the device. For example, you could use a combination of a MAC address and serial number forming the following string for a registration ID: `sn-007-888-abc-mac-a1-b2-c3-d4-e5-f6`. Valid characters are lowercase alphanumeric and dash (-).

Create a DPS enrollment

Create an enrollment to provision one or more devices through DPS.

If you are looking to provision a single IoT Edge device, create an [individual enrollment](#). If you need multiple devices provisioned, follow the steps for creating a [DPS group enrollment](#).

When you create an enrollment in DPS, you have the opportunity to declare an [initial device twin state](#). In the device twin, you can set tags to group devices by any metric you need in your solution, like region, environment, location, or device type. These tags are used to create [automatic deployments](#).

For more information about enrollments in the device provisioning service, see [How to manage device enrollments](#).

Individual enrollment

Create a DPS individual enrollment

💡 Tip

The steps in this article are for the Azure portal, but you can also create individual enrollments using the Azure CLI. For more information, see [az iot dps enrollment](#). As part of the CLI command, use the **edge-enabled** flag to specify that the enrollment is for an IoT Edge device.

1. In the [Azure portal](#), navigate to your instance of IoT Hub device provisioning service.
2. Under **Settings**, select **Manage enrollments**.
3. Select **Add individual enrollment** then complete the following steps to configure the enrollment:
 - a. For **Mechanism**, select **Symmetric Key**.
 - b. Provide a unique **Registration ID** for your device.
 - c. Optionally, provide an **IoT Hub Device ID** for your device. You can use device IDs to target an individual device for module deployment. If you don't provide a device ID, the registration ID is used.
 - d. Select **True** to declare that the enrollment is for an IoT Edge device.
 - e. Optionally, add a tag value to the **Initial Device Twin State**. You can use tags to target groups of devices for module deployment. For example:

JSON

```
{  
  "tags": {  
    "environment": "test"  
  },  
  "properties": {  
    "desired": {}  
  }  
}
```

- f. Select **Save**.

4. Copy the individual enrollment's **Primary Key** value to use when installing the IoT Edge runtime.

Now that an enrollment exists for this device, the IoT Edge runtime can automatically provision the device during installation.

Install IoT Edge

In this section, you prepare your Linux virtual machine or physical device for IoT Edge. Then, you install IoT Edge.

Run the following commands to add the package repository and then add the Microsoft package signing key to your list of trusted keys.

Important

On June 30, 2022 Raspberry Pi OS Stretch was retired from the Tier 1 OS support list. To avoid potential security vulnerabilities update your host OS to Bullseye.

Ubuntu

Installing can be done with a few commands. Open a terminal and run the following commands:

- 22.04:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/22.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

- 20.04:

Bash

```
wget https://packages.microsoft.com/config/ubuntu/20.04/packages-microsoft-prod.deb -O packages-microsoft-prod.deb  
sudo dpkg -i packages-microsoft-prod.deb  
rm packages-microsoft-prod.deb
```

For more information about operating system versions, see [Azure IoT Edge supported platforms](#).

 **Note**

Azure IoT Edge software packages are subject to the license terms located in each package (`usr/share/doc/{package-name}` or the `LICENSE` directory). Read the license terms prior to using a package. Your installation and use of a package constitutes your acceptance of these terms. If you don't agree with the license terms, don't use that package.

Install a container engine

Azure IoT Edge relies on an [OCI](#)-compatible container runtime. For production scenarios, we recommend that you use the Moby engine. The Moby engine is the only container engine officially supported with IoT Edge. Docker CE/EE container images are compatible with the Moby runtime.

Ubuntu

Install the Moby engine.

Bash

```
sudo apt-get update; \
sudo apt-get install moby-engine
```

By default, the container engine doesn't set container log size limits. Over time, this can lead to the device filling up with logs and running out of disk space. However, you can configure your log to show locally, though it's optional. To learn more about logging configuration, see [Production Deployment Checklist](#).

The following steps show you how to configure your container to use [local logging driver](#) as the logging mechanism.

Ubuntu / Debian / RHEL

1. Create or edit the existing Docker [daemon's config file](#)

Bash

```
sudo nano /etc/docker/daemon.json
```

- Set the default logging driver to the `local` logging driver as shown in the example.

JSON

```
{  
  "log-driver": "local"  
}
```

- Restart the container engine for the changes to take effect.

Bash

```
sudo systemctl restart docker
```

Install the IoT Edge runtime

The IoT Edge service provides and maintains security standards on the IoT Edge device. The service starts on every boot and bootstraps the device by starting the rest of the IoT Edge runtime.

Note

Beginning with version 1.2, the [IoT identity service](#) handles identity provisioning and management for IoT Edge and for other device components that need to communicate with IoT Hub.

The steps in this section represent the typical process to install the latest IoT Edge version on a device that has internet connection. If you need to install a specific version, like a pre-release version, or need to install while offline, follow the [Offline or specific version installation](#) steps later in this article.

Tip

If you already have an IoT Edge device running an older version and want to upgrade to the latest release, use the steps in [Update the IoT Edge security](#)

daemon and runtime. Later versions are sufficiently different from previous versions of IoT Edge that specific steps are necessary to upgrade.

Ubuntu

Install the latest version of IoT Edge and the IoT identity service package (if you're not already [up-to-date](#)):

- 22.04:

Bash

```
sudo apt-get update; \
    sudo apt-get install aziot-edge
```

- 20.04:

Bash

```
sudo apt-get update; \
    sudo apt-get install aziot-edge defender-iot-micro-agent-edge
```

The optional `defender-iot-micro-agent-edge` package includes the Microsoft Defender for IoT security micro-agent that provides endpoint visibility into security posture management, vulnerabilities, threat detection, fleet management and more to help you secure your IoT Edge devices. It's recommended to install the micro agent with the Edge agent to enable security monitoring and hardening of your Edge devices. To learn more about Microsoft Defender for IoT, see [What is Microsoft Defender for IoT for device builders](#).

Provision the device with its cloud identity

Once the runtime is installed on your device, configure the device with the information it uses to connect to the device provisioning service and IoT Hub.

Have the following information ready:

- The DPS ID Scope value
- The device Registration ID you created
- Either the Primary Key from an individual enrollment, or a [derived key](#) for devices using a group enrollment.

Create a configuration file for your device based on a template file that is provided as part of the IoT Edge installation.

Ubuntu / Debian / RHEL

Bash

```
sudo cp /etc/aziot/config.toml.edge.template /etc/aziot/config.toml
```

Open the configuration file on the IoT Edge device.

Bash

```
sudo nano /etc/aziot/config.toml
```

1. Find the **Provisioning** section of the file. Uncomment the lines for DPS provisioning with symmetric key, and make sure any other provisioning lines are commented out.

toml

```
# DPS provisioning with symmetric key
[provisioning]
source = "dps"
global_endpoint = "https://global.azure-devices-provisioning.net"
id_scope = "PASTE_YOUR_SCOPE_ID_HERE"

# Uncomment to send a custom payload during DPS registration
# payload = { uri = "PATH_TO_JSON_FILE" }

[provisioning.attestation]
method = "symmetric_key"
registration_id = "PASTE_YOUR_REGISTRATION_ID_HERE"

symmetric_key = { value = "PASTE_YOUR_PRIMARY_KEY_OR_DERIVED_KEY_HERE" }

# auto_reprovisioning_mode = Dynamic
```

2. Update the values of `id_scope`, `registration_id`, and `symmetric_key` with your DPS and device information.

The symmetric key parameter can accept a value of an inline key, a file URI, or a PKCS#11 URI. Uncomment just one symmetric key line, based on which format

you're using. When using an inline key, use a base64-encoded key like the example. When using a file URI, your file should contain the raw bytes of the key.

If you use any PKCS#11 URIs, find the **PKCS#11** section in the config file and provide information about your PKCS#11 configuration.

For more information about provisioning configuration settings, see [Configure IoT Edge device settings](#).

3. Optionally, find the auto reprovisioning mode section of the file. Use the `auto_reprovisioning_mode` parameter to configure your device's reprovisioning behavior. **Dynamic** - Reprovision when the device detects that it may have been moved from one IoT Hub to another. This is the default. **AlwaysOnStartup** - Reprovision when the device is rebooted or a crash causes the daemons to restart. **OnErrorOnly** - Never trigger device reprovisioning automatically. Each mode has an implicit device reprovisioning fallback if the device is unable to connect to IoT Hub during identity provisioning due to connectivity errors. For more information, see [IoT Hub device reprovisioning concepts](#).
4. Optionally, uncomment the `payload` parameter to specify the path to a local JSON file. The contents of the file is sent to DPS as additional data when the device registers. This is useful for [custom allocation](#). For example, if you want to allocate your devices based on an IoT Plug and Play model ID without human intervention.
5. Save and close the file.
6. Apply the configuration changes that you made on the device.

Ubuntu / Debian / RHEL

Bash

```
sudo iotedge config apply
```

Verify successful installation

If the runtime started successfully, you can go into your IoT Hub and start deploying IoT Edge modules to your device.

Individual enrollment

You can verify that the individual enrollment that you created in device provisioning service was used. Navigate to your device provisioning service instance in the Azure portal. Open the enrollment details for the individual enrollment that you created. Notice that the status of the enrollment is **assigned** and the device ID is listed.

Use the following commands on your device to verify that the IoT Edge installed and started successfully.

Check the status of the IoT Edge service.

```
cmd/sh  
sudo iotedge system status
```

Examine service logs.

```
cmd/sh  
sudo iotedge system logs
```

List running modules.

```
cmd/sh  
sudo iotedge list
```

Next steps

The device provisioning service enrollment process lets you set the device ID and device twin tags at the same time as you provision the new device. You can use those values to target individual devices or groups of devices using automatic device management.

Learn how to [Deploy and monitor IoT Edge modules at scale using the Azure portal](#) or [using Azure CLI](#).

Update IoT Edge

Article • 06/13/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024.

As the IoT Edge service releases new versions, update your IoT Edge devices for the latest features and security improvements. This article provides information about how to update your IoT Edge devices when a new version is available.

Two logical components of an IoT Edge device need to be updated if you want to move to a newer version.

- *Security subsystem* - It runs on the device, handles security-based tasks, and starts the modules when the device starts. The *security subsystem* can only be updated from the device itself.
- *IoT Edge runtime* - The IoT Edge runtime is made up of the IoT Edge hub (`edgeHub`) and IoT Edge agent (`edgeAgent`) modules. Depending on how you structure your deployment, the *runtime* can be updated from either the device or remotely.

How to update

Use the sections of this article to update both the security subsystem and runtime containers on a device.

Patch releases

When you upgrade between *patch* releases, for example 1.4.1 to 1.4.2, the update order isn't important. You can upgrade the security subsystem or the runtime containers before or after the other. To update between patch releases:

1. [Update the security subsystem](#)
2. [Update the runtime containers](#)
3. [Verify versions match](#)

You can [troubleshoot](#) the upgrade process at any time.

Major or minor releases

When you upgrade between major or minor releases, for example from 1.4 to 1.5, update both the security subsystem and the runtime containers. Before a release, we test the security subsystem and the runtime container version combination. To update between major or minor product releases:

1. On the device, stop IoT Edge using the command `sudo systemctl stop iotedge` and [uninstall](#).
2. On the device, upgrade your container engine, either [Docker](#) or [Moby](#).
3. On the device, [install IoT Edge](#).

If you're importing an old configuration using `iotedge config import`, then modify the [agent.config] image of the generated `/etc/aziot/config.toml` file to use the 1.4 image for edgeAgent.

For more information, see [Configure IoT Edge device settings](#).

4. In IoT Hub, [update the module deployment](#) to reference the newest system modules.
5. On the device, start the IoT Edge using `sudo iotedge config apply`.

You can [troubleshoot](#) the upgrade process at any time.

Update the security subsystem

The IoT Edge security subsystem includes a set of native components that need to be updated using the package manager on the IoT Edge device.

Check the version of the security subsystem running on your device by using the command `iotedge version`. If you're using IoT Edge for Linux on Windows, you need to SSH into the Linux virtual machine to check the version.

Windows

① Note

Currently, there is no support for IoT Edge running on Windows devices in Windows containers. Use a Linux container to run IoT Edge on Windows.

Then, reapply configuration to ensure system is fully updated.

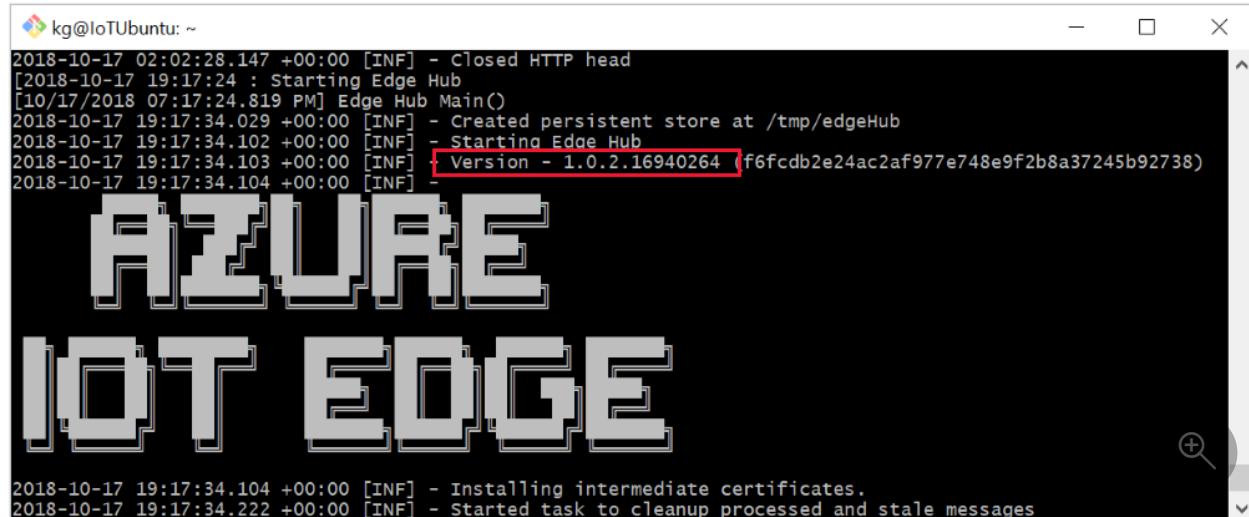
```
Bash
```

```
sudo iotedge config apply
```

Update the runtime containers

The way that you update the IoT Edge agent and IoT Edge hub containers depends on whether you use rolling tags (like 1.5) or specific tags (like 1.5.1) in your deployment.

Check the version of the IoT Edge agent and IoT Edge hub modules currently on your device using the commands `iotedge logs edgeAgent` or `iotedge logs edgeHub`. If you're using IoT Edge for Linux on Windows, you need to SSH into the Linux virtual machine to check the runtime module versions.



```
kg@IoTUbuntu: ~
2018-10-17 02:02:28.147 +00:00 [INF] - Closed HTTP head
[2018-10-17 19:17:24 : Starting Edge Hub
[10/17/2018 07:17:24.819 PM] Edge Hub Main()
2018-10-17 19:17:34.029 +00:00 [INF] - Created persistent store at /tmp/edgeHub
2018-10-17 19:17:34.102 +00:00 [INF] - Starting Edae Hub
2018-10-17 19:17:34.103 +00:00 [INF] - Version - 1.0.2.16940264 (f6fcdb2e24ac2af977e748e9f2b8a37245b92738)
2018-10-17 19:17:34.104 +00:00 [INF] -
[REDACTED]
[AZURE
IOT EDGE]
2018-10-17 19:17:34.104 +00:00 [INF] - Installing intermediate certificates.
2018-10-17 19:17:34.222 +00:00 [INF] - Started task to cleanup processed and stale messages
```

Understand IoT Edge tags

The IoT Edge agent and IoT Edge hub images are tagged with the IoT Edge version that they're associated with. There are two different ways to use tags with the runtime images:

- **Rolling tags** - Use only the first two values of the version number to get the latest image that matches those digits. For example, 1.5 is updated whenever there's a new release to point to the latest 1.5.x version. If the container runtime on your IoT Edge device pulls the image again, the runtime modules are updated to the latest version. Deployments from the Azure portal default to rolling tags. *This approach is suggested for development purposes.*

- **Specific tags** - Use all three values of the version number to explicitly set the image version. For example, 1.5.0 won't change after its initial release. You can declare a new version number in the deployment manifest when you're ready to update. *This approach is suggested for production purposes.*

Update a rolling tag image

If you use rolling tags in your deployment (for example, `mcr.microsoft.com/azureiotedge-hub:1.5`) then you need to force the container runtime on your device to pull the latest version of the image.

Delete the local version of the image from your IoT Edge device. On Windows machines, uninstalling the security subsystem also removes the runtime images, so you don't need to take this step again.

Bash

```
docker rmi mcr.microsoft.com/azureiotedge-hub:1.5
docker rmi mcr.microsoft.com/azureiotedge-agent:1.5
```

You may need to use the force `-f` flag to remove the images.

The IoT Edge service pulls the latest versions of the runtime images and automatically starts them on your device again.

Update a specific tag image

If you use specific tags in your deployment (for example, `mcr.microsoft.com/azureiotedge-hub:1.4`) then all you need to do is update the tag in your deployment manifest and apply the changes to your device.

1. In the IoT Hub in the Azure portal, select your IoT Edge device, and select **Set Modules**.
2. On the **Modules** tab, select **Runtime Settings**.
3. In **Runtime Settings**, update the **Image URI** value in the **Edge Agent** section with the desired version. For example, `mcr.microsoft.com/azureiotedge-agent:1.5` Don't select **Apply** yet.
4. Select the **Edge Hub** tab and update the **Image URI** value with the same desired version. For example, `mcr.microsoft.com/azureiotedge-hub:1.5`.

5. Select **Apply** to save changes.
6. Select **Review + create**, review the deployment as seen in the JSON file, and select **Create**.

Update partner module URIs

If you use partner modules, update your module deployments with image URIs provided by the partner. Contact the [IoT Edge module publisher](#) to obtain the updated container image URI. Update your device configurations with the new image URI provided by the publisher.

1. Sign in to the [Azure portal](#) and navigate to your IoT Hub.
2. On the left pane, select **Devices** under the **Device management** menu.
3. Select the IoT Edge device that uses the partner module from the list.
4. On the upper bar, select **Set Modules**.
5. Choose the IoT Edge partner module that you want to update with the new image URI.
6. Update the **Image URI** value with the new image URI provided by the publisher.
7. Select **Apply** to save changes.
8. Select **Review + create**, review the deployment as seen in the JSON file, and select **Create**.

Verify versions match

1. On your device, use `iotedge version` to check the security subsystem version. The output includes the major, minor, and revision version numbers. For example, `iotedge 1.4.2`.
2. In your device deployment runtime settings, verify the `edgeHub` and `edgeAgent` image URI versions match the major and minor version of the security subsystem. If the security subsystem version is 1.4.2, the image versions would be 1.4. For example, `mcr.microsoft.com/azureiotedge-hub:1.4` and `mcr.microsoft.com/azureiotedge-agent:1.4`.

Note

Update the IoT Edge security subsystem and runtime containers to the same supported release version. While mismatched versions are supported, we haven't tested all version combinations.

To find the latest version of Azure IoT Edge, see [Azure IoT Edge releases](#).

Troubleshooting

You can view logs of your system at any time by running the following commands from your device.

- Start troubleshooting using the `check` command. It runs a collection of configuration and connectivity tests for common issues.

```
Bash
```

```
sudo iotedge check --verbose
```

- To view the status of the IoT Edge system, run:

```
Bash
```

```
sudo iotedge system status
```

- To view host component logs, run:

```
Bash
```

```
sudo iotedge system logs
```

- To check for recurring issues reported with `edgeAgent` and `edgeHub`, run:

Be sure to replace `<module>` with your own module name. If there are no issues, you see no output.

```
Bash
```

```
sudo iotedge logs <module>
```

For more information, see [Troubleshoot your IoT Edge device](#).

Next steps

View the latest [Azure IoT Edge releases](#).

Stay up-to-date with recent updates and announcements in the [Internet of Things blog](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Manage IoT Edge certificates

Article • 04/09/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

All IoT Edge devices use certificates to create secure connections between the runtime and any modules running on the device. IoT Edge devices functioning as gateways use these same certificates to connect to their downstream devices, too.

Note

The term *root CA* used throughout this article refers to the topmost authority's certificate in the certificate chain for your IoT solution. You don't need to use the certificate root of a syndicated certificate authority, or the root of your organization's certificate authority. Often, it's actually an intermediate CA certificate.

Prerequisites

- You should be familiar with the concepts in [Understand how Azure IoT Edge uses certificates](#), in particular how IoT Edge uses certificates.
- An IoT Edge device.

If you don't have an IoT Edge device set up, you can create one in an Azure virtual machine. Follow the steps in one of these quickstart articles to [Create a virtual Linux device](#) or [Create a virtual Windows device](#).

- Ability to edit the IoT Edge configuration file `config.toml` following the [configuration template ↗](#).
- If your `config.toml` isn't based on the template, open the [template ↗](#) and use the commented guidance to add configuration sections following the structure of the template.

- If you have a new IoT Edge installation that hasn't been configured, copy the template to initialize the configuration. Don't use this command if you have an existing configuration. It overwrites the file.

```
Bash
```

```
sudo cp /etc/aziot/config.toml.edge.template /etc/aziot/config.toml
```

Format requirements

Tip

- A certificate can be encoded in a binary representation called DER (Distinguished Encoding Rules), or a textual representation called PEM (Privacy Enhanced Mail). The PEM format has a -----BEGIN CERTIFICATE----- header followed by the base64-encoded DER followed by an -----END CERTIFICATE----- footer.
- Similar to the certificate, the private key can be encoded in binary DER or textual representation PEM.
- Because PEM is delineated, it's also possible to construct a PEM that combines both the `CERTIFICATE` and `PRIVATE KEY` sequentially in the same file.
- Lastly, the certificate and private key can be encoded together in a binary representation called *PKCS#12*, that's encrypted with an optional password.

File extensions are arbitrary and you need to run the `file` command or view the file verify the type. In general, files use the following extension conventions:

- `.cer` is a certificate in DER or PEM form.
- `.pem` is either a certificate, private key, or both in PEM form.
- `.pfx` is a *PKCS#12* file.

IoT Edge requires the certificate and private key to be:

- PEM format
- Separate files
- In most cases, with the full chain

If you get a `.pfx` file from your PKI provider, it's likely the certificate and private key encoded together in one file. Verify it's a *PKCS#12* file type by using the `file` command.

You can convert a PKCS#12 `.pfx` file to PEM files using the [openssl pkcs12 command](#).

If your PKI provider provides a `.cer` file, it may contain the same certificate as the `.pfx`, or it might be the PKI provider's issuing (root) certificate. To verify, inspect the file with the `openssl x509` command. If it's the issuing certificate:

- If it's in DER (binary) format, convert it to PEM with `openssl x509 -in cert.cer -out cert.pem`.
- Use the PEM file as the trust bundle. For more information about the trust bundle, see the next section.

Important

Your PKI infrastructure should support RSA-2048 bit keys and EC P-256 keys. For example, your EST servers should support these key types. You can use other key types, but we only test RSA-2048 bit keys and EC P-256 keys.

Permission requirements

The following table lists the file and directory permissions required for the IoT Edge certificates. The preferred directory for the certificates is `/var/aziot/certs/` and `/var/aziot/secrets/` for keys.

[\[+\] Expand table](#)

File or directory	Permissions	Owner
<code>/var/aziot/certs/</code> certificates directory	<code>drwxr-xr-x (755)</code>	aziotcs
Certificate files in <code>/var/aziot/certs/</code>	<code>-wr-r--r-- (644)</code>	aziotcs
<code>/var/aziot/secrets/</code> keys directory	<code>drwx----- (700)</code>	aziotks
Key files in <code>/var/aziot/secrets/</code>	<code>-wr----- (600)</code>	aziotks

To create the directories, set the permissions, and set the owner, run the following commands:

Bash

```
# If the certificate and keys directories don't exist, create, set ownership, and set permissions
sudo mkdir -p /var/aziot/certs
sudo chown aziotcs:aziotcs /var/aziot/certs
```

```

sudo chmod 755 /var/aziot/certs

sudo mkdir -p /var/aziot/secrets
sudo chown aziotks:aziotks /var/aziot/secrets
sudo chmod 700 /var/aziot/secrets

# Give aziotcs ownership to certificates
# Read and write for aziotcs, read-only for others
sudo chown -R aziotcs:aziotcs /var/aziot/certs
sudo find /var/aziot/certs -type f -name "*.*" -exec chmod 644 {} \;

# Give aziotks ownership to private keys
# Read and write for aziotks, no permission for others
sudo chown -R aziotks:aziotks /var/aziot/secrets
sudo find /var/aziot/secrets -type f -name "*.*" -exec chmod 600 {} \;

# Verify permissions of directories and files
sudo ls -Rla /var/aziot

```

The output of the list with the correct ownership and permission is similar to the following output:

Output

```

azureUser@vm:/var/aziot$ sudo ls -Rla /var/aziot
/var/aziot:
total 16
drwxr-xr-x  4 root      root      4096 Dec 14 00:16 .
drwxr-xr-x 15 root      root      4096 Dec 14 00:15 ..
drwxr-xr-x  2 aziotcs  aziotcs  4096 Jan 14 00:31 certs
drwx-----  2 aziotks  aziotks  4096 Jan 23 17:23 secrets

/var/aziot/certs:
total 20
drwxr-xr-x  2 aziotcs  aziotcs  4096 Jan 14 00:31 .
drwxr-xr-x  4 root      root      4096 Dec 14 00:16 ..
-rw-r--r--  1 aziotcs  aziotcs 1984 Jan 14 00:24 azure-iot-test-
only.root.ca.cert.pem
-rw-r--r--  1 aziotcs  aziotcs 5887 Jan 14 00:27 iot-edge-device-ca-
devicename-full-chain.cert.pem

/var/aziot/secrets:
total 16
drwx-----  2 aziotks  aziotks  4096 Jan 23 17:23 .
drwxr-xr-x  4 root      root      4096 Dec 14 00:16 ..
-rw-----  1 aziotks  aziotks 3243 Jan 14 00:28 iot-edge-device-ca-
devicename.key.pem

```

Manage trusted root CA (trust bundle)

Using a self-signed certificate authority (CA) certificate as a root of trust with IoT Edge and modules is known as *trust bundle*. The trust bundle is available for IoT Edge and modules to communicate with servers. To configure the trust bundle, specify its file path in the IoT Edge configuration file.

1. Get the root CA certificate from a PKI provider.
2. Check that the certificate meets the [format requirements](#).
3. Copy the PEM file and give IoT Edge's certificate service access. For example, with `/var/aziot/certs` directory:

Bash

```
# Make the directory if doesn't exist
sudo mkdir /var/aziot/certs -p

# Change cert directory user and group ownership to aziotcs and set
# permissions
sudo chown aziotcs:aziotcs /var/aziot/certs
sudo chmod 755 /var/aziot/certs

# Copy certificate into certs directory
sudo cp root-ca.pem /var/aziot/certs

# Give aziotcs ownership to certificate and set read and write
# permission for aziotcs, read-only for others
sudo chown aziotcs:aziotcs /var/aziot/certs/root-ca.pem
sudo chmod 644 /var/aziot/certs/root-ca.pem
```

4. In the IoT Edge configuration file `config.toml`, find the **Trust bundle cert** section. If the section is missing, you can copy it from the configuration template file.

 **Tip**

If the config file doesn't exist on your device yet, then use `/etc/aziot/config.toml.edge.template` as a template to create one.

5. Set the `trust_bundle_cert` key to the certificate file location.

toml

```
trust_bundle_cert = "file:///var/aziot/certs/root-ca.pem"
```

6. Apply the configuration.

```
Bash
```

```
sudo iotedge config apply
```

Install root CA to OS certificate store

Installing the certificate to the trust bundle file makes it available to container modules but not to host modules like Azure Device Update or Defender. If you use host level components or run into other TLS issues, also install the root CA certificate to the operating system certificate store:

```
EFLOW / RHEL
```

```
Bash
```

```
sudo cp /var/aziot/certs/my-root-ca.pem /etc/pki/ca-trust/source/anchors/my-root-ca.pem.crt
```

```
sudo update-ca-trust
```

Import certificate and private key files

IoT Edge can use existing certificates and private key files to authenticate or attest to Azure, issue new module server certificates, and authenticate to EST servers. To install them:

1. Check the certificate and private key files meet the [format requirements](#).
2. Copy the PEM file to the IoT Edge device where IoT Edge modules can have access.

For example, the `/var/aziot/` directory.

```
Bash
```

```
# If the certificate and keys directories don't exist, create, set ownership, and set permissions
sudo mkdir -p /var/aziot/certs
sudo chown aziotcs:aziotcs /var/aziot/certs
sudo chmod 755 /var/aziot/certs

sudo mkdir -p /var/aziot/secrets
sudo chown aziotks:aziotks /var/aziot/secrets
sudo chmod 700 /var/aziot/secrets

# Copy certificate and private key into the correct directory
```

```
sudo cp my-cert.pem /var/aziot/certs  
sudo cp my-private-key.pem /var/aziot/secrets
```

3. Grant ownership to IoT Edge's certificate service `aziotcs` and key service `aziotks` to the certificate and private key, respectively.

Bash

```
# Give aziotcs ownership to certificate  
# Read and write for aziotcs, read-only for others  
sudo chown aziotcs:aziotcs /var/aziot/certs/my-cert.pem  
sudo chmod 644 /var/aziot/certs/my-cert.pem  
  
# Give aziotks ownership to private key  
# Read and write for aziotks, no permission for others  
sudo chown aziotks:aziotks /var/aziot/secrets/my-private-key.pem  
sudo chmod 600 /var/aziot/secrets/my-private-key.pem
```

4. In `config.toml`, find the relevant section for the type of the certificate to configure. For example, you can search for the keyword `cert`.

5. Using the example from the configuration template, configure the device identity certificate or Edge CA files. The example pattern is:

toml

```
cert = "file:///var/aziot/certs/my-cert.pem"  
pk = "file:///var/aziot/secrets/my-private-key.pem"
```

6. Apply the configuration

Bash

```
sudo iotedge config apply
```

To prevent errors when certificates expire, remember to manually update the files and configuration before certificate expiration.

Example: Use device identity certificate files from PKI provider

Request a TLS client certificate and a private key from your PKI provider.

Device identity certificate requirements:

- Standard client certificate extensions: extendedKeyUsage = clientAuth keyUsage = critical, digitalSignature
- Key identifiers to help distinguish between issuing CAs with the same CN for CA certificate rotation.
 - subjectKeyIdentifier = hash
 - authorityKeyIdentifier = keyid(always,issuer(always)

Ensure that the common name (CN) matches the IoT Edge device ID registered with IoT Hub or registration ID with DPS. For example, in the following device identity certificate, `Subject: CN = my-device` is the important field that must match.

Example device identity certificate:

```
Output

Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 48 (0x30)
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: CN = myPkiCA
    Validity
      Not Before: Jun 28 21:27:30 2022 GMT
      Not After : Jul 28 21:27:30 2022 GMT
    Subject: CN = my-device
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public-Key: (2048 bit)
      Modulus:
        00:ad:b0:63:1f:48:19:9e:c4:9d:91:d1:b0:b0:e5:
        ...
        80:58:63:6d:ab:56:9f:90:4e:3f:dd:df:74:cf:86:
        04:af
      Exponent: 65537 (0x10001)
    X509v3 extensions:
      X509v3 Basic Constraints:
        CA:FALSE
      X509v3 Key Usage:
        Digital Signature
      X509v3 Extended Key Usage:
        TLS Web Client Authentication
      X509v3 Subject Key Identifier:
        C7:C2:DC:3C:53:71:B8:42:15:D5:6C:4B:5C:03:C2:2A:C5:98:82:7E
      X509v3 Authority Key Identifier:
        keyid:6E:57:C7:FC:FE:50:09:75:FA:D9:89:13:CB:D2:CA:F2:28:EF:9B:F6

      Signature Algorithm: ecdsa-with-SHA256
      30:45:02:20:3c:d2:db:06:3c:d7:65:b7:22:fe:df:9e:11:5b:
      ...
      eb:da:fc:f1:6a:bf:31:63:db:5a:16:02:70:0f:cf:c8:e2
```

```
-----BEGIN CERTIFICATE-----  
MIICdTCCAhuAgIBAgIBMDAKBggqhkJOPQQDAjAXMRUwEwYDVQQDDAx1c3RFeGFT  
...  
354RWw+eLOpQSkTqXxzjmfw/kV0OAQIhANvRmyCQVb8zLPtqdOVRkuva/PFqvzFj  
21oWAnAPz8ji  
-----END CERTIFICATE-----
```

💡 Tip

To test without access to certificate files provided by a PKI, see [Create demo certificates to test device features](#) to generate a short-lived non-production device identity certificate and private key.

Configuration example when provisioning with IoT Hub:

```
toml  
  
[provisioning]  
source = "manual"  
# ...  
[provisioning.authentication]  
method = "x509"  
  
identity_cert = "file:///var/aziot/device-id.pem"  
identity_pk = "file:///var/aziot/device-id.key.pem"
```

Configuration example when provisioning with DPS:

```
toml  
  
[provisioning]  
source = "dps"  
# ...  
[provisioning.attestation]  
method = "x509"  
registration_id = "my-device"  
  
identity_cert = "file:///var/aziot/device-id.pem"  
identity_pk = "file:///var/aziot/device-id.key.pem"
```

Overhead with manual certificate management can be risky and error-prone. For production, using IoT Edge with automatic certificate management is recommended.

Manage Edge CA

Edge CA has two different modes:

- *Quickstart* is the default behavior. Quickstart is for testing and **not** suitable for production.
- *Production* mode requires you provide your own source for Edge CA certificate and private key.

Quickstart Edge CA

To help with getting started, IoT Edge automatically generates an **Edge CA certificate** when started for the first time by default. This self-signed certificate is only meant for development and testing scenarios, not production. By default, the certificate expires after 90 days. Expiration can be configured. This behavior is referred to as *quickstart Edge CA*.

Quickstart Edge CA enables `edgeHub` and other IoT Edge modules to have a valid server certificate when IoT Edge is first installed with no configuration. The certificate is needed by `edgeHub` because modules or downstream devices [need to establish secure communication channels](#). Without the quickstart Edge CA, getting started would be significantly harder because you'd need to provide a valid server certificate from a PKI provider or with tools like `openssl`.

ⓘ Important

Never use the quickstart Edge CA for production because the locally generated certificate in it isn't connected to a PKI.

The security of a certificates-based identity derives from a well-operated PKI (the infrastructure) in which the certificate (a document) is only a component. A well-operated PKI enables definition, application, management, and enforcements of security policies to include but not limited to certificates issuance, revocation, and lifecycle management.

Customize lifetime for quickstart Edge CA

To configure the certificate expiration to something other than the default 90 days, add the value in days to the **Edge CA certificate (Quickstart)** section of the config file.

```
toml
```

```
[edge_ca]
```

```
auto_generated_edge_ca_expiry_days = 180
```

Delete the contents of the `/var/lib/aziot/certd/certs` and `/var/lib/aziot/keyd/keys` folders to remove any previously generated certificates then apply the configuration.

Renew quickstart Edge CA

By default, IoT Edge automatically renews the quickstart Edge CA certificate when at 80% of the certificate lifetime. For example, if a certificate has a 90 day lifetime, IoT Edge automatically regenerates the Edge CA certificate at 72 days from issuance.

To change the auto-renewal logic, add the following settings to the *Edge CA certificate* section in `config.toml`. For example:

```
toml
```

```
[edge_ca.auto_renew]
rotate_key = true
threshold = "70%"
retry = "2%"
```

Edge CA in production

Once you move into a production scenario, or you want to create a gateway device, you can no longer use the quickstart Edge CA.

One option is to provide your own certificates and manage them manually. However, to avoid the risky and error-prone manual certificate management process, use an EST server whenever possible.

⊗ Caution

The common name (CN) of Edge CA certificate can't match device hostname parameter defined in the device's configuration file `config.toml` or the device ID registered in IoT Hub.

Plan for Edge CA renewal

When the Edge CA certificate renews, all the certificates it issued like module server certificates are regenerated. To give the modules new server certificates, IoT Edge restarts all modules when Edge CA certificate renews.

To minimize potential negative effects of module restarts, plan to renew the Edge CA certificate at a specific time (for example, `threshold = "10d"`) and notify dependents of the solution about the downtime.

Example: use Edge CA certificate files from PKI provider

Request the following files from your PKI provider:

- The PKI's root CA certificate
- An issuing/CA certificate and associated private key

For the issuing CA certificate to become Edge CA, it must have these extensions:

text

```
subjectKeyIdentifier = hash
authorityKeyIdentifier = keyid:always
basicConstraints = critical, CA:TRUE, pathlen:0
keyUsage = critical, digitalSignature, keyCertSign
```

Example of the result Edge CA certificate:

Bash

```
openssl x509 -in my-edge-ca-cert.pem -text
```

Output

```
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number: 4098 (0x1002)
    Signature Algorithm: sha256WithRSAEncryption
    Issuer: CN = myPkiCA
    Validity
      Not Before: Aug 27 00:00:50 2022 GMT
      Not After : Sep 26 00:00:50 2022 GMT
    Subject: CN = my-edge-ca.ca
    Subject Public Key Info:
      Public Key Algorithm: rsaEncryption
      RSA Public-Key: (4096 bit)
      Modulus:
        00:e1:cb:9c:c0:41:d2:ee:5d:8b:92:f9:4e:0d:3e:
        ...
        25:f5:58:1e:8c:66:ab:d1:56:78:a5:9c:96:eb:01:
        e4:e3:49
      Exponent: 65537 (0x10001)
    X509v3 extensions:
```

```
X509v3 Subject Key Identifier:  
    FD:64:48:BB:41:CE:C1:8A:8A:50:9B:2B:2D:6E:1D:E5:3F:86:7D:3E  
X509v3 Authority Key Identifier:  
  
keyid:9F:E6:D3:26:EE:2F:D7:84:09:63:84:C8:93:72:D5:13:06:8E:7F:D1  
    X509v3 Basic Constraints: critical  
        CA:TRUE, pathlen:0  
    X509v3 Key Usage: critical  
        Digital Signature, Certificate Sign  
Signature Algorithm: sha256WithRSAEncryption  
    20:c9:34:41:a3:a4:8e:7c:9c:6e:17:f5:a6:6f:e5:fc:6e:59:  
    ...  
    7c:20:5d:e5:51:85:4c:4d:f7:f8:01:84:87:27:e3:76:65:47:  
    9e:6a:c3:2e:1a:f0:dc:9d  
-----BEGIN CERTIFICATE-----  
MIICdTCCAhugAwIBAgIBMDAKBggqhkjOPQQDAjAXMRUwEwYDVQQDDAxlc3RFeGFT  
...  
354RWw+eLOpQSkTqXxzjmfw/kV0OAQIhANvRmyCQVb8zLPtqdOVRkuva/PFqvzFj  
21oWAnAPz8ji  
-----END CERTIFICATE-----
```

Once you receive the latest files, [update the trust bundle](#):

```
toml  
  
trust_bundle_cert = "file:///var/aziot/root-ca.pem"
```

Then, configure IoT Edge to use the certificate and private key files:

```
toml  
  
[edge_ca]  
cert = "file:///var/aziot/my-edge-ca-cert.pem"  
pk = "file:///var/aziot/my-edge-ca-private-key.key.pem"
```

If you've used any other certificates for IoT Edge on the device before, delete the files in `/var/lib/aziot/certd/certs` and the private keys associated with certificates (*not all keys*) in `/var/lib/aziot/keyd/keys`. IoT Edge recreates them with the new CA certificate you provided.

This approach requires you to manually update the files as certificate expires. To avoid this issue, consider using EST for automatic management.

Automatic certificate management with EST server

IoT Edge can interface with an [Enrollment over Secure Transport \(EST\) server](#) for automatic certificate issuance and renewal. Using EST is recommended for production as it replaces the need for manual certificate management, which can be risky and error-prone. It can be configured globally and overridden for each certificate type.

In this scenario, the bootstrap certificate and private key are expected to be long-lived and potentially installed on the device during manufacturing. IoT Edge uses the bootstrap credentials to authenticate to the EST server for the initial request to issue an identity certificate for subsequent requests and for authentication to DPS or IoT Hub.

1. Get access to an EST server. If you don't have an EST server, use one of the following options to start testing:
 - Create a test EST server using the steps in [Tutorial: Configure Enrollment over Secure Transport Server for Azure IoT Edge](#).
 - Microsoft partners with GlobalSign to [provide a demo account](#).
2. In the IoT Edge device configuration file `config.toml`, configure the path to a trusted root certificate that IoT Edge uses to validate the EST server's TLS certificate. This step is optional if the EST server has a publicly trusted root TLS certificate.

```
toml

[cert_issuance.est]
trusted_certs = [
    "file:///var/aziot/root-ca.pem",
]
```

3. Provide a default URL for the EST server. In `config.toml`, add the following section with the URL of the EST server:

```
toml

[cert_issuance.est.urls]
default = "https://example.org/.well-known/est"
```

4. To configure the EST certificate for authentication, add the following section with the path to the certificate and private key:

```
toml

[cert_issuance.est.auth]
bootstrap_identity_cert = "file:///var/aziot/my-est-id-bootstrap-
```

```
cert.pem"
bootstrap_identity_pk = "file:///var/aziot/my-est-id-bootstrap-
pk.key.pem"

[cert_issuance.est.identity_auto_renew]
rotate_key = true
threshold = "80%"
retry = "4%"
```

5. Apply the configuration changes.

Bash

```
sudo iotedge config apply
```

The settings in `[cert_issuance.est.identity_auto_renew]` are covered in the next section.

Username and password authentication

If authentication to EST server using certificate isn't possible, you can use a shared secret or username and password instead.

toml

```
[cert_issuance.est.auth]
username = "username"
password = "password"
```

Configure auto-renew parameters

Instead of manually managing the certificate files, IoT Edge has the built-in ability to get and renew certificates before expiry. Certificate renewal requires an issuance method that IoT Edge can manage. Enrollment over Secure Transport (EST) server is one issuance method, but IoT Edge can also automatically [renew the quickstart CA by default](#).

Certificate renewal is configured per type of certificate.

1. In `config.toml`, find the relevant section for the type of the certificate to configure. For example, you can search for the keyword `auto_renew`.
2. Using the example from the configuration template, configure the device identity certificate, Edge CA, or EST identity certificates. The example pattern is:

toml

```
[REPLACE_WITH_CERT_TYPE]
# ...
method = "est"
# ...

[REPLACE_WITH_CERT_TYPE.auto_renew]
rotate_key = true
threshold = "80%"
retry = "4%"
```

3. Apply the configuration

Bash

```
sudo iotege config apply
```

The following table lists what each option in `auto_renew` does:

[] [Expand table](#)

Parameter	Description
<code>rotate_key</code>	Controls if the private key should be rotated when IoT Edge renews the certificate.
<code>threshold</code>	Sets when IoT Edge should start renewing the certificate. It can be specified as: - Percentage: integer between <code>0</code> and <code>100</code> followed by <code>%</code> . Renewal starts relative to the certificate lifetime. For example, when set to <code>80%</code> , a certificate that is valid for 100 days begins renewal at 20 days before its expiry. - Absolute time: integer followed by <code>min</code> (minutes) or <code>day</code> (days). Renewal starts relative to the certificate expiration time. For example, when set to <code>4day</code> for four days or <code>10min</code> for 10 minutes, the certificate begins renewing at that time before expiry. To avoid unintentional misconfiguration where the <code>threshold</code> is bigger than the certificate lifetime, we recommend using <i>percentage</i> instead whenever possible.
<code>retry</code>	controls how often renewal should be retried on failure. Like <code>threshold</code> , it can similarly be specified as a <i>percentage</i> or <i>absolute time</i> using the same format.

Example: renew device identity certificate automatically with EST

To use EST and IoT Edge for automatic device identity certificate issuance and renewal, which is recommended for production, IoT Edge must provision as part of a [DPS CA-based enrollment group](#). For example:

toml

```
## DPS provisioning with X.509 certificate
[provisioning]
source = "dps"
# ...
[provisioning.attestation]
method = "x509"
registration_id = "my-device"

[provisioning.attestation.identity_cert]
method = "est"
common_name = "my-device"

[provisioning.attestation.identity_cert.auto_renew]
rotate_key = true
threshold = "80%"
retry = "4%"
```

Automatic renewal for Edge CA must be enabled when issuance method is set to EST. Edge CA expiration must be avoided as it breaks many IoT Edge functionalities. If a situation requires total control over Edge CA certificate lifecycle, use the [manual Edge CA management method](#) instead.

Don't use EST or `auto_renew` with other methods of provisioning, including manual X.509 provisioning with IoT Hub and DPS with individual enrollment. IoT Edge can't update certificate thumbprints in Azure when a certificate is renewed, which prevents IoT Edge from reconnecting.

Example: automatic Edge CA management with EST

Use EST automatic Edge CA issuance and renewal for production. Once EST server is configured, you can use the global setting, or override it similar to this example:

```
toml

[edge_ca]
method = "est"

common_name = "my-edge-CA"
url = "https://ca.example.org/.well-known/est"

bootstrap_identity_cert = "file:///var/aziot/my-est-id-bootstrap-cert.pem"
bootstrap_identity_pk = "file:///var/aziot/my-est-id-bootstrap-pk.key.pem"

[edge_ca.auto_renew]
rotate_key = true
threshold = "90%"
retry = "2%"
```

Module server certificates

Edge Daemon issues module server and identity certificates for use by Edge modules. It remains the responsibility of Edge modules to renew their identity and server certificates as needed.

Renewal

Server certificates may be issued off the Edge CA certificate. Regardless of the issuance method, these certificates must be renewed by the module. If you develop a custom module, you must implement the renewal logic in your module.

The `edgeHub` module supports a certificate renewal feature. You can configure the `edgeHub` module server certificate renewal using the following environment variables:

- **ServerCertificateRenewAfterInMs**: Sets the duration in milliseconds when the `edgeHub` server certificate is renewed irrespective of certificate expiry time.
- **MaxCheckCertExpiryInMs**: Sets the duration in milliseconds when `edgeHub` service checks the `edgeHub` server certificate expiration. If the variable is set, the check happens irrespective of certificate expiry time.

For more information about the environment variables, see [EdgeHub and EdgeAgent environment variables ↗](#).

Changes in 1.2 and later

- The **Device CA certificate** was renamed as **Edge CA certificate**.
- The **workload CA certificate** was deprecated. Now the IoT Edge security manager generates the IoT Edge hub `edgeHub` server certificate directly from the Edge CA certificate, without the intermediate workload CA certificate between them.
- The default config file has a new name and location, from `/etc/iotedge/config.yaml` to `/etc/aziot/config.toml` by default. The `iotedge config import` command can be used to help migrate configuration information from the old location and syntax to the new one.

Next steps

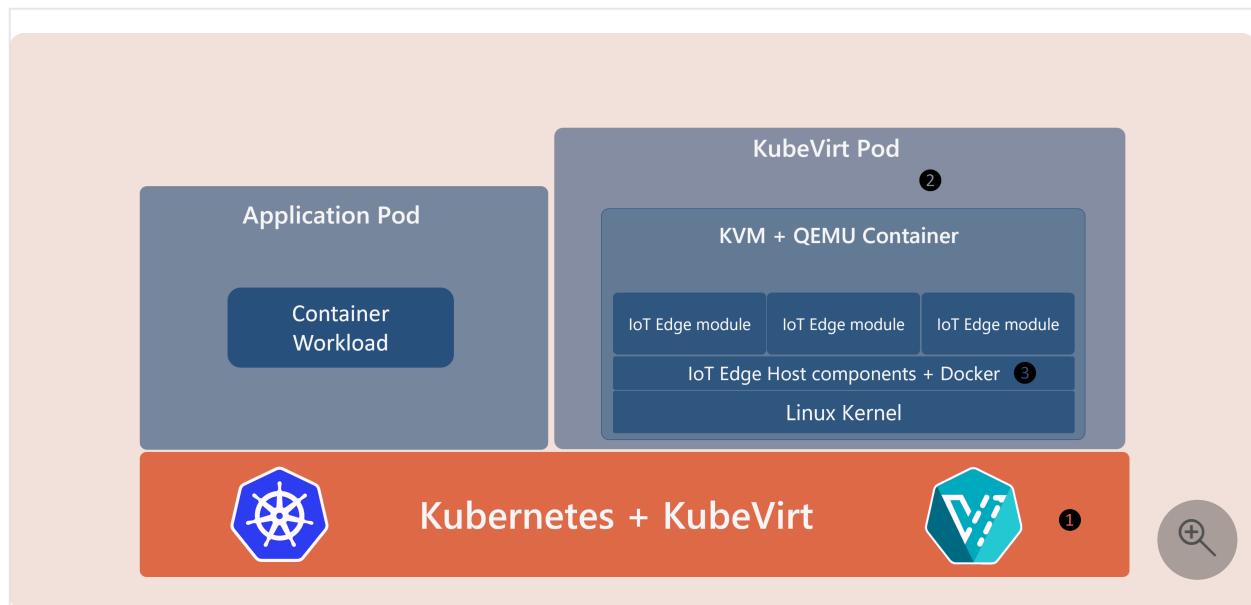
Installing certificates on an IoT Edge device is a necessary step before deploying your solution in production. Learn more about how to [Prepare to deploy your IoT Edge solution in production](#).

How to install IoT Edge on Kubernetes

Article • 05/01/2024

IoT Edge can be installed on Kubernetes by using [KubeVirt](#) technology. KubeVirt is an open source, Cloud Native Computing Foundation (CNCF) project that offers a Kubernetes virtualization API and runtime to define and manage virtual machines.

Architecture



[\[+\] Expand table](#)

Note	Description
1	Install KubeVirt Custom Resource Definitions (CRDs) into the Kubernetes cluster. Like the Kubernetes cluster, management and updates to KubeVirt components are outside the purview of IoT Edge.
2	A KubeVirt <code>VirtualMachine</code> custom resource is used to define a Virtual Machine with required resources and base operating system. A running <i>instance</i> of this resource is created in a Kubernetes Pod using KVM and QEMU technologies. If your Kubernetes node itself is a Virtual Machine, you'll need to enable Nested Virtualization to use KubeVirt.
3	The environment inside the QEMU container is just like an OS environment. IoT Edge and its dependencies (like the Docker container engine) can be setup using standard installation instructions or a cloud-init script.

Sample

A functional sample for running IoT Edge on Azure Kubernetes Service (AKS) using KubeVirt is available at <https://aka.ms/iotedge-kubevirt>.

Run Azure IoT Edge on Ubuntu Virtual Machines by using Bicep

Article • 06/10/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The Azure IoT Edge runtime is what turns a device into an IoT Edge device. The runtime can be deployed on devices as small as a Raspberry Pi or as large as an industrial server. Once a device is configured with the IoT Edge runtime, you can start deploying business logic to it from the cloud.

To learn more about how the IoT Edge runtime works and what components are included, see [Understand the Azure IoT Edge runtime and its architecture](#).

Deploy from Azure CLI

You can't deploy a remote Bicep file. Save a copy of the [Bicep file](#) locally as `main.bicep`.

1. Ensure that you installed the Azure CLI iot extension with:

```
Azure CLI
```

```
az extension add --name azure-iot
```

2. Next, if you're using Azure CLI on your desktop, start by logging in:

```
Azure CLI
```

```
az login
```

3. If you have multiple subscriptions, select the subscription you'd like to use:

- a. List your subscriptions:

Azure CLI

```
az account list --output table
```

b. Copy the SubscriptionID field for the subscription you'd like to use.

c. Set your working subscription with the ID that you copied:

Azure CLI

```
az account set -s <SubscriptionId>
```

4. Create a new resource group (or specify an existing one in the next steps):

Azure CLI

```
az group create --name IoTEdgeResources --location westus2
```

5. Create a new virtual machine:

To use an **authenticationType** of `password`, see the following example:

Azure CLI

```
az deployment group create \
--resource-group IoTEdgeResources \
--template-file "main.bicep" \
--parameters dnsLabelPrefix='my-edge-vm1' \
--parameters deviceConnectionString=$(az iot hub device-identity
connection-string show --device-id <REPLACE_WITH_DEVICE-NAME> --hub-
name <REPLACE-WITH-HUB-NAME> -o tsv) \
--parameters authenticationType='password' \
--parameters adminUsername='<REPLACE_WITH_USERNAME>' \
--parameters adminPasswordOrKey="<REPLACE_WITH_SECRET_PASSWORD>"
```

To authenticate with an SSH key, you may do so by specifying an **authenticationType** of `sshPublicKey`, then provide the value of the SSH key in the `adminPasswordOrKey` parameter. For example:

Azure CLI

```
#Generate the SSH Key
ssh-keygen -m PEM -t rsa -b 4096 -q -f ~/.ssh/iotedge-vm-key -N ""

#Create a VM using the iotedge-vm-deploy script
az deployment group create \
--resource-group IoTEdgeResources \
```

```
--template-file "main.bicep" \
--parameters dnsLabelPrefix='my-edge-vm1' \
--parameters deviceConnectionString=$(az iot hub device-identity
connection-string show --device-id <REPLACE_WITH_DEVICE-NAME> --hub-
name <REPLACE_WITH-HUB-NAME> -o tsv) \
--parameters authenticationType='sshPublicKey' \
--parameters adminUsername='<REPLACE_WITH_USERNAME>' \
--parameters adminPasswordOrKey="$(cat ~/ssh/iotedge-vm-key.pub)"
```

- Verify that the deployment completed successfully. A virtual machine resource should be deployed into the selected resource group. Take note of the machine name, this should be in the format `vm-0000000000000000`. Also, take note of the associated **DNS Name**, which should be in the format `<dnsLabelPrefix>.cloudapp.azure.com`.

The **DNS Name** can be obtained from the JSON-formatted output of the previous step, within the **outputs** section as part of the **public SSH** entry. The value of this entry can be used to SSH into to the newly deployed machine.

Bash

```
"outputs": {
    "public SSH": {
        "type": "String",
        "value": "ssh <adminUsername>@<DNS_Name>"
    }
}
```

The **DNS Name** can also be obtained from the **Overview** section of the newly deployed virtual machine within the Azure portal.

The screenshot shows the Azure portal's Resource Groups section. A specific virtual machine named "vm-vto3qhfbaqx34" is selected. The "Overview" tab is active, displaying details such as Status (Running), Location (West US 2), Subscription (ca-pdecarlo-demo-test), and more. On the right side, under the "Networking" section, the "Virtual network/subnet" field is listed as "vnet-vto3qhfbaqx34/subnet-vto3qhfbaqx34". Below it, the "DNS name" field is highlighted with a red box and shows the value "my-edge-vm99.westus2.cloudapp.azure.com".

- If you want to SSH into this VM after setup, use the associated **DNS Name** with the command: `ssh <adminUsername>@<DNS_Name>`

Next steps

Now that you have an IoT Edge device provisioned with the runtime installed, you can [deploy IoT Edge modules](#).

If you are having problems with the IoT Edge runtime installing properly, check out the [troubleshooting](#) page.

To update an existing installation to the newest version of IoT Edge, see [Update the IoT Edge security daemon and runtime](#).

If you'd like to open up ports to access the VM through SSH or other inbound connections, refer to the Azure Virtual Machines documentation on [opening up ports and endpoints to a Linux VM](#).

Feedback

Was this page helpful?



[Provide product feedback ↗](#)

Run Azure IoT Edge on Ubuntu Virtual Machines

Article • 06/03/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The Azure IoT Edge runtime is what turns a device into an IoT Edge device. The runtime can be deployed on devices as small as a Raspberry Pi or as large as an industrial server. Once a device is configured with the IoT Edge runtime, you can start deploying business logic to it from the cloud.

To learn more about how the IoT Edge runtime works and what components are included, see [Understand the Azure IoT Edge runtime and its architecture](#).

This article lists the steps to deploy an Ubuntu virtual machine with the Azure IoT Edge runtime installed and configured using a presupplied device connection string. The deployment is accomplished using a [cloud-init](#) based [Azure Resource Manager template](#) maintained in the [iotedge-vm-deploy](#) project repository.

On first boot, the virtual machine [installs the latest version of the Azure IoT Edge runtime via cloud-init](#). It also sets a supplied connection string before the runtime starts, allowing you to easily configure and connect the IoT Edge device without the need to start an SSH or remote desktop session.

Deploy using Deploy to Azure Button

The [Deploy to Azure Button](#) allows for streamlined deployment of [Azure Resource Manager templates](#) maintained on GitHub. This section demonstrates usage of the Deploy to Azure Button contained in the [iotedge-vm-deploy](#) project repository.

1. You will deploy an Azure IoT Edge enabled Linux VM using the iotedge-vm-deploy Azure Resource Manager template. To begin, select the following button:



2. On the newly launched window, fill in the available form fields:

The screenshot shows the Microsoft Azure 'Custom deployment' interface. At the top, there's a navigation bar with 'Home > Custom deployment'. Below it, a section titled 'Deploy from a custom template' shows a 'Customized template' with 5 resources. There are three buttons: 'Edit template', 'Edit parameters', and 'Visualize'. The 'Project details' section asks to select a subscription and resource group. The 'Subscription' dropdown is set to 'Visual Studio Enterprise'. The 'Resource group' dropdown shows '(New) learn-azure-iot-edge' with a 'Create new' option. The 'Instance details' section contains several input fields: 'Region' (East US), 'Dns Label Prefix' (myedgedevice-03951a19), 'Admin Username' (micah), 'Device Connection String' (HostName=myedgedevice-03951a19.azure-devices.net;DeviceId=m...), 'Vm Size' (Standard_DS1_v2), 'Ubuntu OS Version' (20_04-lts), 'Authentication Type' (selected 'Password'), 'Admin Password Or Key' (redacted), and 'Allow Ssh' (set to 'true').

[+] Expand table

Field	Description
Subscription	The active Azure subscription to deploy the virtual machine into.
Resource group	An existing or newly created Resource Group to contain the virtual machine and its associated resources.
Region	The geographic region to deploy the virtual machine into, this value defaults to the location of the selected Resource Group.
DNS Label Prefix	A required value of your choosing that is used to prefix the hostname of the virtual machine.
Admin Username	A username that is provided root privileges on deployment.
Device Connection String	A device connection string for a device that was created within your intended IoT Hub.

Field	Description
VM Size	The size of the virtual machine to be deployed.
Ubuntu OS Version	The version of the Ubuntu OS to be installed on the base virtual machine.
Authentication Type	Choose sshPublicKey or password depending on your preference.
Admin Password or Key	The value of the SSH Public Key or the value of the password depending on the choice of Authentication Type.

Select [Next : Review + create](#) to review the terms and select [Create](#) to begin the deployment.

- Verify that the deployment completed successfully. A virtual machine resource is deployed into the selected resource group. Take note of the machine name, this should be in the format `vm-00000000000000`. Also, take note of the associated **DNS Name**, which should be in the format `<dnsLabelPrefix>`.

`<location>.cloudapp.azure.com.`

The **DNS Name** can be obtained from the **Overview** section of the newly deployed virtual machine within the Azure portal.

Setting	Value
Azure Spot	N/A
Public IP address	13.66.229.133
Private IP address	10.0.0.4
Public IP address (IPv6)	-
Private IP address (IPv6)	-
Virtual network/subnet	vnet-vto3qhfboax34/subnet-vto3qhfboax34
DNS name	my-edge-vm99.westus2.cloudapp.azure.com
Scale Set	N/A

- If you want to SSH into this VM after setup, use the associated **DNS Name** with the command: `ssh <adminUsername>@<DNS_Name>`

Deploy from Azure CLI

- Ensure that you installed the Azure CLI iot extension with:

```
Azure CLI
az extension add --name azure-iot
```

- Next, if you're using Azure CLI on your desktop, start by logging in:

Azure CLI

```
az login
```

3. If you have multiple subscriptions, select the subscription you'd like to use:

a. List your subscriptions:

Azure CLI

```
az account list --output table
```

b. Copy the SubscriptionID field for the subscription you'd like to use.

c. Set your working subscription with the ID that you copied:

Azure CLI

```
az account set -s <SubscriptionId>
```

4. Create a new resource group (or specify an existing one in the next steps):

Azure CLI

```
az group create --name IoTEdgeResources --location westus2
```

5. Create a new virtual machine:

To use an **authenticationType** of `password`, see the following example:

Azure CLI

```
az deployment group create \
--resource-group IoTEdgeResources \
--template-uri "https://raw.githubusercontent.com/Azure/iotedge-vm-deploy/main/edgeDeploy.json" \
--parameters dnsLabelPrefix='my-edge-vm1' \
--parameters adminUsername='<REPLACE_WITH_USERNAME>' \
--parameters deviceConnectionString=$(az iot hub device-identity connection-string show --device-id <REPLACE_WITH_DEVICE-NAME> --hub-name <REPLACE-WITH-HUB-NAME> -o tsv) \
--parameters authenticationType='password' \
--parameters adminPasswordOrKey="<REPLACE_WITH_SECRET_PASSWORD>"
```

To authenticate with an SSH key, specify an **authenticationType** of `sshPublicKey`, then provide the value of the SSH key in the **adminPasswordOrKey** parameter. See

the following example:

Azure CLI

```
#Generate the SSH Key
ssh-keygen -m PEM -t rsa -b 4096 -q -f ~/.ssh/iotedge-vm-key -N ""

#Create a VM using the iotedge-vm-deploy script
az deployment group create \
--resource-group IoTEdgeResources \
--template-uri "https://raw.githubusercontent.com/Azure/iotedge-vm-
deploy/main/edgeDeploy.json" \
--parameters dnsLabelPrefix='my-edge-vm1' \
--parameters adminUsername='<REPLACE_WITH_USERNAME>' \
--parameters deviceConnectionString=$(az iot hub device-identity
connection-string show --device-id <REPLACE_WITH_DEVICE-NAME> --hub-
name <REPLACE-WITH-HUB-NAME> -o tsv) \
--parameters authenticationType='sshPublicKey' \
--parameters adminPasswordOrKey="$(cat ~/ssh/iotedge-vm-key.pub)"
```

6. Verify that the deployment completed successfully. A virtual machine resource should be deployed into the selected resource group. Take note of the machine name, this should be in the format `vm-00000000000000`. Also, take note of the associated **DNS Name**, which should be in the format `<dnsLabelPrefix>`.
`<location>.cloudapp.azure.com.`

The **DNS Name** can be obtained from the JSON-formatted output of the previous step, within the **outputs** section as part of the **public SSH** entry. The value of this entry can be used to SSH into to the newly deployed machine.

Bash

```
"outputs": {
  "public SSH": {
    "type": "String",
    "value": "ssh <adminUsername>@<DNS_Name>"
  }
}
```

The **DNS Name** can also be obtained from the **Overview** section of the newly deployed virtual machine within the Azure portal.

The screenshot shows the Azure portal interface for a virtual machine named 'vm-vto3qhfbaqx34'. The left sidebar has sections for Overview, Activity log, Tags, Diagnose and solve problems, Settings, Networking, and Connect. The main content area displays various details about the VM, including its resource group ('IoTEdgeResources'), status ('Running'), location ('West US 2'), subscription ('ca-pedcarlo-demo-test'), computer name ('vm-vto3qhfbaqx34'), operating system ('Linux (ubuntu 18.04)'), size ('Standard DS1 v2 (1 vcpus, 3.5 GiB memory)'), and tags ('Click here to add tags'). On the right, there are columns for Azure Spot (N/A), Public IP address (13.66.229.133), Private IP address (10.0.0.4), Public IP address (IPv6) (-), Private IP address (IPv6) (-), Virtual network/subnet (vnet-vto3qhfbaqx34/subnet-vto3qhfbaqx34), and Scale Set (N/A). The 'DNS name' field is highlighted with a red box and contains the value 'my-edge-vm99.westus2.cloudapp.azure.com'.

7. If you want to SSH into this VM after setup, use the associated **DNS Name** with the command: `ssh <adminUsername>@<DNS_Name>`

Next steps

Now that you have an IoT Edge device provisioned with the runtime installed, you can [deploy IoT Edge modules](#).

If you are having problems with the IoT Edge runtime installing properly, check out the [troubleshooting](#) page.

To update an existing installation to the newest version of IoT Edge, see [Update the IoT Edge security daemon and runtime](#).

If you'd like to open up ports to access the VM through SSH or other inbound connections, refer to the Azure Virtual Machines documentation on [opening up ports and endpoints to a Linux VM](#).

Configure IoT Edge module build options

Article • 05/31/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The *module.json* file controls how modules are built and deployed. IoT Edge module Visual Studio and Visual Studio Code projects include the *module.json* file. The file contains IoT Edge module configuration details including the version and platform that is used when building an IoT Edge module.

module.json settings

The *module.json* file includes the following settings:

 Expand table

Setting	Description
image.repository	The repository of the module.
image.tag.version	The version of the module.
image.tag.platforms	A list of supported platforms and their corresponding dockerfile. Each entry is a platform key and dockerfile pair <code><platform key>:<dockerfile></code> .
image.buildOptions	The build arguments used when running <code>docker build</code> .
image.contextPath	The context path used when running <code>docker build</code> . By default, it's the current folder of the <i>module.json</i> file. If your Docker build needs files not included in the current folder such as a reference to an external package or project, set the contextPath to the root path of all necessary files. Verify the files are copied in the dockerfile.
language	The programming language of the module.

For example, the following *module.json* file is for a C# IoT Edge module:

JSON

```
{  
    "$schema-version": "0.0.1",  
    "description": "",  
    "image": {  
        "repository": "localhost:5000/edgemodule",  
        "tag": {  
            "version": "0.0.1",  
            "platforms": {  
                "amd64": "./Dockerfile.amd64",  
                "amd64.debug": "./Dockerfile.amd64.debug",  
                "arm32v7": "./Dockerfile.arm32v7",  
                "arm32v7.debug": "./Dockerfile.arm32v7.debug",  
                "arm64v8": "./Dockerfile.arm64v8",  
                "arm64v8.debug": "./Dockerfile.arm64v8.debug",  
                "windows-amd64": "./Dockerfile.windows-amd64"  
            }  
        },  
        "buildOptions": [ "--add-host=docker:10.180.0.1" ],  
        "contextPath": "./"  
    },  
    "language": "csharp"  
}
```

Once the module is built, the final tag of the image is combined with both version and platform as `<repository>:<version>-<platform key>`. For this example, the image tag for `amd64.debug` is `localhost:5000/csharpmod:0.0.1-amd64.debug`.

Next step

[Understand the requirements and tools for developing IoT Edge modules](#)

How to configure container create options for IoT Edge modules

Article • 02/21/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The `createOptions` parameter in the deployment manifest enables you to configure the module containers at runtime. This parameter expands your control over the modules and allows for tasks like allowing or restricting the module's access to the host device's resources, or configuring networking.

IoT Edge modules are implemented as Docker-compatible containers on your IoT Edge device. Docker offers many options for creating containers, and those options apply to IoT Edge modules, too. For more information, see [Docker container create options](#).

Format create options

The IoT Edge deployment manifest accepts create options formatted as JSON. For example, take the create options that are automatically included for every edgeHub module:

JSON

```
"createOptions": {  
    "HostConfig": {  
        "PortBindings": {  
            "5671/tcp": [  
                {  
                    "HostPort": "5671"  
                }  
            ],  
            "8883/tcp": [  
                {  
                    "HostPort": "8883"  
                }  
            ],  
            "443/tcp": [  
                {  
                    "HostPort": "443"  
                }  
            ]  
        }  
    }  
}
```

```
        {
          "HostPort": "443"
        }
      ]
    }
}
```

This edgeHub example uses the `HostConfig.PortBindings` parameter to map exposed ports on the container to a port on the host device.

If you use the [Azure IoT Edge](#) extension for Visual Studio or Visual Studio Code, you can write the create options in JSON format in the `deployment.template.json` file. Then, when you use the extension to build the IoT Edge solution or generate the deployment manifest, it will stringify the JSON for you in the format that the IoT Edge runtime expects. For example:

JSON

```
"createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":
[{\\"HostPort\":\"5671\"}],\\\"8883/tcp\\\":
[{\\"HostPort\":\"8883\"}],\\\"443/tcp\\\": [{\\"HostPort\":\"443\"}]}}}"
```

ⓘ Important

The Azure IoT Edge Visual Studio Code extension is in [maintenance mode](#). The `iotedgedev` tool is the recommended tool for developing IoT Edge modules.

One tip for writing create options is to use the `docker inspect` command. As part of your development process, run the module locally using `docker run <container name>`. Once you have the module working the way you want it, run `docker inspect <container name>`. This command outputs the module details in JSON format. Find the parameters that you configured, and copy the JSON. For example:

```

kg@ubuntu109:~$ sudo docker inspect edgeHub
[{"Id": "18b9acf4ead875dfcd5ae36a3a88ae3a05561ffd44a0ee3653d28743df0a6bbe",
 "Created": "2020-03-17T21:39:40.295364912Z",
 "Path": "/bin/sh",
 "Args": [
   "-c",
   "echo \"\$date --utc +\"%Y-%m-%d %H:%M:%S %:z\" Starting Edge Hub\" & exec /usr/bin/dotnet Microsoft.Azure.Devices.Edge.Hub.Service.dll"
 ],
 "State": {
   ...
 },
 "Image": "sha256:0723d28bf40b6844a2dee51533cb3abaabc572d0b3ea1ec3533a0c143fe18b18",
 "Config": {
   "ConfigPath": "/var/lib/docker/containers/18b9acf4ead875dfcd5ae36a3a88ae3a05561ffd44a0ee3653d28743df0a6bbe/resolv.conf",
   "HostnamePath": "/var/lib/docker/containers/18b9acf4ead875dfcd5ae36a3a88ae3a05561ffd44a0ee3653d28743df0a6bbe/hostname",
   "HostPath": "/var/lib/docker/containers/18b9acf4ead875dfcd5ae36a3a88ae3a05561ffd44a0ee3653d28743df0a6bbe/hosts",
   "LogPath": "/var/lib/docker/containers/18b9acf4ead875dfcd5ae36a3a88ae3a05561ffd44a0ee3653d28743df0a6bbe/18b9acf4ead875dfcd5ae36a3a88ae3a05561ffd44a0ee3653d28743df0a6bbe/18b9acf4ead875dfcd5ae36a3a88ae3a05561ffd44a0ee3653d28743df0a6bbe/hosts.log",
   "Name": "/edgeHub",
   "Platform": "linux",
   "HostConfig": {
     "Binds": [
       "/var/run/iotedge/workload.sock:/var/run/iotedge/workload.sock"
     ],
     "LogConfig": {
       "Type": "json-file",
       "Config": {}
     }
   },
   "NetworkMode": "default",
   "PortBindings": {
     "443/tcp": [
       {
         "HostIp": "",
         "HostPort": "443"
       }
     ],
     "5671/tcp": [
       {
         "HostIp": "",
         "HostPort": "5671"
       }
     ],
     "8883/tcp": [
       {
         "HostIp": "",
         "HostPort": "8883"
       }
     ]
   },
   "RestartPolicy": {
     "Name": "",
     "MaximumRetryCount": 0
   },
   "AutoRemove": false,
 }
}

```

Common scenarios

Container create options enable many scenarios, but here are some that come up most often when building IoT Edge solutions:

- Give modules access to host storage
- Map host port to module port
- Restrict module memory and CPU usage
- GPU-optimize an IoT Edge module

Map host port to module port

If your module needs to communicate with a service outside of the IoT Edge solution, and isn't using message routing to do so, then you need to map a host port to a module port.

Tip

This port mapping is not required for module-to-module communication on the same device. If module A needs to query an API hosted on module B, it can do so without any port mapping. Module B needs to expose a port in its dockerfile, for example: `EXPOSE 8080`. Then module A can query the API using module B's name, for example: `http://ModuleB:8080/api`.

First, make sure that a port inside the module is exposed to listen for connections. You can do this using an [EXPOSE](#) instruction in the dockerfile. For example, `EXPOSE 8080`. The expose instruction defaults to TCP protocol if not specified, or you can specify UDP.

Then, use the `PortBindings` setting in the `HostConfig` group of the [Docker container create options](#) to map the exposed port in the module to a port on the host device. For example, if you exposed port 8080 inside the module and want to map that to port 80 of the host device, the create options in the template.json file would look like the following example:

JSON

```
"createOptions": {  
  "HostConfig": {  
    "PortBindings": {  
      "8080/tcp": [  
        {  
          "HostPort": "80"  
        }  
      ]  
    }  
  }  
}
```

Once stringified for the deployment manifest, the same configuration would look like the following example:

JSON

```
"createOptions": "{\"HostConfig\":{\"PortBindings\":{\"8080/tcp\": [\"HostPort\":\"80\"]}}}"
```

Restrict module memory and CPU usage

You can declare how much of the host resources a module can use. This control is helpful to ensure that one module can't consume too much memory or CPU usage and prevent other processes from running on the device. You can manage these settings with [Docker container create options](#) in the `HostConfig` group, including:

- **Memory**: Memory limit in bytes. For example, 268435456 bytes = 256 MB.
- **MemorySwap**: Total memory limit (memory + swap). For example, 536870912 bytes = 512 MB.
- **NanoCpus**: CPU quota in units of 10^{-9} (1 billionth) CPUs. For example, 250000000 nanocpus = 0.25 CPU.

In the template.json format, these values would look like the following example:

JSON

```
"createOptions": {  
    "HostConfig": {  
        "Memory": 268435456,  
        "MemorySwap": 536870912,  
        "NanoCpus": 250000000  
    }  
}
```

Once stringified for the final deployment manifest, these values would look like the following example:

JSON

```
"createOptions": "{\"HostConfig\":  
  {\"Memory\":268435456, \"MemorySwap\":536870912, \"CpuPeriod\":25000}}"
```

GPU-optimize an IoT Edge module

If you're running your IoT Edge module on a GPU-optimized virtual machine, you can enable an IoT Edge module to connect to your GPU as well. To do this with an existing module, add some specifications to your `createOptions`:

JSON

```
{"HostConfig": {"DeviceRequests": [{"Count": -1, "Capabilities":  
  [["gpu"]]}]}}
```

To confirm these settings were successfully added, use the Docker inspect command to see the new setting in a JSON printout.

Bash

```
sudo docker inspect <YOUR-MODULE-NAME>
```

To learn more about how your device and virtual machine connect to a GPU, see [Configure, connect, and verify an IoT Edge module for a GPU](#).

Next steps

For more examples of create options in action, see the following IoT Edge samples:

- [Custom Vision and Azure IoT Edge on a Raspberry Pi 3 ↗](#)
- [Azure IoT Edge blob storage sample ↗](#)

Give Azure IoT Edge modules access to a device's local storage

Article • 06/10/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

IoT Edge modules can use storage on the host IoT Edge device itself for improved reliability, especially when operating offline.

Configure system modules to use persistent storage

By default, IoT Edge system modules, IoT Edge agent and IoT Edge hub, store state in the ephemeral file system of their container instance. This state is lost when the container instance is recycled, for example, when module image version or `createOptions` is updated.

For production scenarios, use a persistent storage location on the host filesystem to store system module state. Doing so improves solution robustness and cloud message delivery guarantees.

To set up system modules to use persistent storage:

1. For both IoT Edge hub and IoT Edge agent, add an environment variable called `StorageFolder` that points to a directory in the module.
2. For both IoT Edge hub and IoT Edge agent, add binds to connect a local directory on the host machine to a directory in the module. For example:

Image URI *

mcr.microsoft.com/azureiotedge-agent:1.5

Schema version

1.1

Image Pull Policy

Always

Environment Variables

Environment variables provide supplemental information to a module facilitating the configuration process.

NAME	TYPE	VALUE
StorageFolder	Text	<ModuleStoragePath>
<i>Variable name</i>	Text	<i>Variable value</i>

Container Create Options

Create options direct the creation of the IoT Edge module Docker container. [View all options](#).

```

1  {
2    "HostConfig": {
3      "Binds": [
4        "<HostStoragePath>:<ModuleStoragePath>"
5      ]
6    }
7 }
```

Apply **Cancel**

Replace `<HostStoragePath>` and `<ModuleStoragePath>` with your host and module storage path. Both values must be an absolute path and `<HostStoragePath>` must exist.

You can configure the local storage directly in the deployment manifest. For example, if you want to map the following storage paths:

[] Expand table

Module	Host storage path	Module storage path
edgeAgent	/srv/edgeAgent	/tmp/edgeAgent
edgeHub	/srv/edgeHub	/tmp/edgeHub

Your deployment manifest would be similar to the following:

JSON

```
"systemModules": {
    "edgeAgent": {
        "env": {
            "StorageFolder": {
                "value": "/tmp/edgeAgent"
            }
        },
        "settings": {
            "image": "mcr.microsoft.com/azureiotedge-agent:1.5",
            "createOptions": "{\"HostConfig\":{\"Binds\": [
                \"/srv/edgeAgent:/tmp/edgeAgent\"
            ]}}"
        },
        "type": "docker"
    },
    "edgeHub": {
        "env": {
            "StorageFolder": {
                "value": "/tmp/edgeHub"
            }
        },
        "restartPolicy": "always",
        "settings": {
            "image": "mcr.microsoft.com/azureiotedge-hub:1.5",
            "createOptions": "{\"HostConfig\":{\"Binds\": [
                \"/srv/edgeHub:/tmp/edgeHub\",
                \"PortBindings\": {
                    \"443/tcp\": [
                        {\"HostPort\": \"443\"}
                    ],
                    \"5671/tcp\": [
                        {\"HostPort\": \"5671\"}
                    ],
                    \"8883/tcp\": [
                        {\"HostPort\": \"8883\"}
                    ]
                }
            ]}}"
        },
        "status": "running",
        "type": "docker"
    }
}
```

ⓘ Note

If you are using a snap installation, ensure you choose a host storage path that is accessible to the snaps. For example, `$HOME/snap/azure-iot-edge/current/modules/`.

Automatic host system permissions management

On version 1.4 and newer, there's no need for manually setting ownership or permissions for host storage backing the `StorageFolder`. Permissions and ownership are automatically managed by the system modules during startup.

ⓘ Note

Automatic permission management of host bound storage only applies to system modules, IoT Edge agent and Edge hub. For custom modules, manual management of permissions and ownership of bound host storage is required if the custom module container isn't running as `root` user.

Link module storage to device storage for custom modules

If your custom module requires access to persistent storage on the host file system, use the module's create options to bind a storage folder in module container to a folder on the host machine. For example:

JSON

```
{  
  "HostConfig": {  
    "Mounts": [  
      {  
        "Target": "<ModuleStoragePath>",  
        "Source": "<HostStoragePath>",  
        "Type": "bind",  
        "ReadOnly": false  
      }  
    ]  
  }  
}
```

Replace `<HostStoragePath>` and `<ModuleStoragePath>` with your host and module storage path; both values must be an absolute path. Refer to the [Docker Engine Mount specification](#) for option details.

Host system permissions

Make sure that the user profile your module is using has the required read, write, and execute permissions to the host system directory. By default, containers run as `root` user that already has the required permissions. But your module's Dockerfile might specify use of a non-root user in which case host storage permissions must be manually configured.

There are several ways to manage directory permissions on Linux systems, including using `chown` to change the directory owner and then `chmod` to change the permissions. For example to allow host storage access to a module running as non-root user ID 1000, use the following commands:

Bash

```
sudo chown 1000 <HostStoragePath>
sudo chmod 700 <HostStoragePath>
```

Encrypted data in module storage

When modules invoke the IoT Edge daemon's workload API to encrypt data, the encryption key is derived using the module ID and module's generation ID. A generation ID is used to protect secrets if a module is removed from the deployment and then another module with the same module ID is later deployed to the same device. You can view a module's generation ID using the Azure CLI command [az iot hub module-identity show](#).

If you want to share files between modules across generations, they must not contain any secrets or they will fail to be decrypted.

Next steps

For an additional example of accessing host storage from a module, see [Store data at the edge with Azure Blob Storage on IoT Edge](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Debug Azure IoT Edge modules using Visual Studio Code

Article • 02/21/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article shows you how to use Visual Studio Code to debug IoT Edge modules in multiple languages. On your development computer, you can use Visual Studio Code to attach and debug your module in a local or remote module container.

This article includes steps for two IoT Edge development tools.

- *Azure IoT Edge Dev Tool* command-line tool (CLI). This tool is preferred for development.
- *Azure IoT Edge tools for Visual Studio Code* extension. The extension is in [maintenance mode](#).

Use the tool selector button at the beginning of this article to select the tool version.

Visual Studio Code supports writing IoT Edge modules in the following programming languages:

- C# and C# Azure Functions
- C
- Python
- Node.js
- Java

Azure IoT Edge supports the following device architectures:

- AMD64
- ARM32v7
- ARM64

For more information about supported operating systems, languages, and architectures, see [Language and architecture support](#).

You can also use a Windows development computer and debug modules in a Linux container using IoT Edge for Linux on Windows (EFLOW). For more information about using EFLOW for developing modules, see [Tutorial: Develop IoT Edge modules with Linux containers using IoT Edge for Linux on Windows](#).

If you aren't familiar with the debugging capabilities of Visual Studio Code, see [Visual Studio Code debugging](#).

Prerequisites

You can use a computer or a virtual machine running Windows, macOS, or Linux as your development machine. On Windows computers, you can develop either Windows or Linux modules. To develop Linux modules, use a Windows computer that meets the [requirements for Docker Desktop](#).

To install the required tools for development and debugging, complete the [Develop Azure IoT Edge modules using Visual Studio Code](#) tutorial.

Install [Visual Studio Code](#)

To debug your module on a device, you need:

- An active IoT Hub with at least one IoT Edge device.
- A physical IoT Edge device or a virtual device. To create a virtual device in Azure, follow the steps in the quickstart for [Linux](#).
- A custom IoT Edge module. To create a custom module, follow the steps in the [Develop Azure IoT Edge modules using Visual Studio Code](#) tutorial.

Debug a module with the IoT Edge runtime

In each module folder, there are several Docker files for different container types. Use any of the files that end with the extension `.debug` to build your module for testing.

When you debug modules using this method, your modules are running on top of the IoT Edge runtime. The IoT Edge device and your Visual Studio Code can be on the same machine, or more typically, Visual Studio Code is on the development machine and the IoT Edge runtime and modules are running on another physical machine. To debug from Visual Studio Code, you must:

- Set up your IoT Edge device, build your IoT Edge modules with the `.debug` Dockerfile, and then deploy to the IoT Edge device.

- Update `launch.json` so that Visual Studio Code can attach to the process in a container on the remote machine. You can find this file in the `.vscode` folder in your workspace, and it updates each time you add a new module that supports debugging.
- Use Remote SSH debugging to attach to the container on the remote machine.

Build and deploy your module to an IoT Edge device

In Visual Studio Code, open the `deployment.debug.template.json` deployment manifest file. The [deployment manifest](#) describes the modules to be configured on the targeted IoT Edge device. Before deployment, you need to update your Azure Container Registry credentials and your module images with the proper `createOptions` values. For more information about `createOption` values, see [How to configure container create options for IoT Edge modules](#).

1. If you're using an Azure Container Registry to store your module image, add your credentials to the `edgeAgent > settings > registryCredentials` section in `deployment.debug.template.json`. Replace `myacr` with your own registry name in both places and provide your password and **Login server** address. For example:

```
JSON

"modulesContent": {
  "$edgeAgent": {
    "properties.desired": {
      "schemaVersion": "1.1",
      "runtime": {
        "type": "docker",
        "settings": {
          "minDockerVersion": "v1.25",
          "loggingOptions": "",
          "registryCredentials": {
            "myacr": {
              "username": "myacr",
              "password": "<your.azure_container_registry_password>",
              "address": "myacr.azurecr.io"
            }
          }
        }
      },
      ...
    }
  }
},
```

2. Add or replace the following stringified content to the `createOptions` value for each system (`edgeHub` and `edgeAgent`) and custom module (for example, `filtermodule`) listed. Change the values if necessary.

JSON

```
"createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}\"}
```

For example, the *filtermodule* configuration should be similar to:

JSON

```
"filtermodule": {
  "version": "1.0",
  "type": "docker",
  "status": "running",
  "restartPolicy": "always",
  "settings": {
    "image": "myacr.azurecr.io/filtermodule:0.0.1-amd64",
    "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}\"}"
  }
}
```

Sign in to Docker

Provide your container registry credentials to Docker so that it can push your container image to storage in the registry.

1. Sign in to Docker with the Azure Container Registry credentials that you saved after creating the registry.

Bash

```
docker login -u <Azure Container Registry username> -p <Azure Container Registry password> <Azure Container Registry login server>
```

You may receive a security warning recommending the use of `--password-stdin`. While that's a recommended best practice for production scenarios, it's outside the scope of this tutorial. For more information, see the [docker login](#) reference.

2. Sign in to the Azure Container Registry. You may need to [Install Azure CLI](#) to use the `az` command. This command asks for your user name and password found in your container registry in **Settings > Access keys**.

Azure CLI

```
az acr login -n <Azure Container Registry name>
```

💡 Tip

If you get logged out at any point in this tutorial, repeat the Docker and Azure Container Registry sign in steps to continue.

Build module Docker image

Use the module's Dockerfile to [build ↗](#) the module Docker image.

Bash

```
docker build --rm -f "<DockerFilePath>" -t <ImageNameAndTag> "<ContextPath>"
```

For example, to build the image for the local registry or an Azure Container Registry, use the following commands:

Bash

```
# Build the image for the local registry

docker build --rm -f "./modules/filtermodule/Dockerfile.amd64.debug" -t
localhost:5000/filtermodule:0.0.1-amd64 "./modules/filtermodule"

# Or build the image for an Azure Container Registry

docker build --rm -f "./modules/filtermodule/Dockerfile.amd64.debug" -t
myacr.azurecr.io/filtermodule:0.0.1-amd64 "./modules/filtermodule"
```

Push module Docker image

[Push ↗](#) your module image to the local registry or a container registry.

```
docker push <ImageName>
```

For example:

Bash

```
# Push the Docker image to the local registry

docker push localhost:5000/filtermodule:0.0.1-amd64
```

```
# Or push the Docker image to an Azure Container Registry
az acr login --name myacr
docker push myacr.azurecr.io/filtermodule:0.0.1-amd64
```

Deploy the module to the IoT Edge device

Use the [IoT Edge Azure CLI set-modules](#) command to deploy the modules to the Azure IoT Hub. For example, to deploy the modules defined in the `deployment.debug.template.json` file to IoT Hub `my-iot-hub` for the IoT Edge device `my-device`, use the following command:

Azure CLI

```
az iot edge set-modules --hub-name my-iot-hub --device-id my-device --
  content ./deployment.debug.template.json --login "HostName=my-iot-hub.azure-
  devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=
  <SharedAccessKey>"
```

Tip

You can find your IoT Hub shared access key in the Azure portal in your IoT Hub > **Security settings** > **Shared access policies** > `iothubowner`.

Debug your module

To debug modules on a remote device, you can use Remote SSH debugging in Visual Studio Code.

To enable Visual Studio Code remote debugging, install the [Remote Development extension](#). For more information about Visual Studio Code remote debugging, see [Visual Studio Code Remote Development](#).

For details on how to use Remote SSH debugging in Visual Studio Code, see [Remote Development using SSH](#)

In the Visual Studio Code Debug view, select the debug configuration file for your module. By default, the `.debug` Dockerfile, module's container `createOptions` settings, and the `launch.json` file use `localhost`.

Select **Start Debugging** or select **F5**. Select the process to attach to. In the Visual Studio Code Debug view, you see variables in the left panel.

Debug using Docker Remote SSH

The Docker and Moby engines support SSH connections to containers allowing you to debug in Visual Studio Code connected to a remote device. You need to meet the following prerequisites before you can use this feature.

Remote SSH debugging prerequisites may be different depending on the language you are using. The following sections describe the setup for .NET. For information on other languages, see [Remote Development using SSH](#) for an overview. Details about how to configure remote debugging are included in debugging sections for each language in the Visual Studio Code documentation.

Configure Docker SSH tunneling

1. Follow the steps in [Docker SSH tunneling](#) to configure SSH tunneling on your development computer. SSH tunneling requires public/private key pair authentication and a Docker context defining the remote device endpoint.
2. Connecting to Docker requires root-level privileges. Follow the steps in [Manage docker as a non-root user](#) to allow connection to the Docker daemon on the remote device. When you finish debugging, you may want to remove your user from the Docker group.
3. In Visual Studio Code, use the Command Palette (Ctrl+Shift+P) to issue the *Docker Context: Use* command to activate the Docker context pointing to the remote machine. This command causes both Visual Studio Code and Docker CLI to use the remote machine context.

💡 Tip

All Docker commands use the current context. Remember to change context back to *default* when you are done debugging.

4. To verify the remote Docker context is active, list the running containers on the remote device:

```
Bash
```

```
docker ps
```

The output should list the containers running on the remote device similar:

Output

```
PS C:\> docker ps
CONTAINER ID        IMAGE
COMMAND             CREATED          STATUS           PORTS
NAMES
a317b8058786      myacr.azurecr.io/filtermodule:0.0.1-amd64
"dotnet filtermodule..."   24 hours ago   Up 6 minutes
filtermodule
d4d949f8dfb9      mcr.microsoft.com/azureiotedge-hub:1.5
"/bin/sh -c 'echo \"$...\" 24 hours ago   Up 6 minutes  0.0.0.0:443->443/tcp, :::443->443/tcp, 0.0.0.0:5671->5671/tcp, :::5671->5671/tcp, 0.0.0.0:8883->8883/tcp, :::8883->8883/tcp, 1883/tcp  edgeHub
1f0da9cfe8e8      mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.0  "/bin/sh -c 'echo \"$...\" 24 hours ago   Up 6 minutes
tempSensor
66078969d843      mcr.microsoft.com/azureiotedge-agent:1.5
"/bin/sh -c 'exec /a...'  24 hours ago   Up 6 minutes
edgeAgent
```

5. In the .vscode directory, add a new configuration to **launch.json** by opening the file in Visual Studio Code. Select **Add configuration** then choose the matching remote attach template for your module. For example, the following configuration is for .NET Core. Change the value for the **-H** parameter in **PipeArgs** to your device DNS name or IP address.

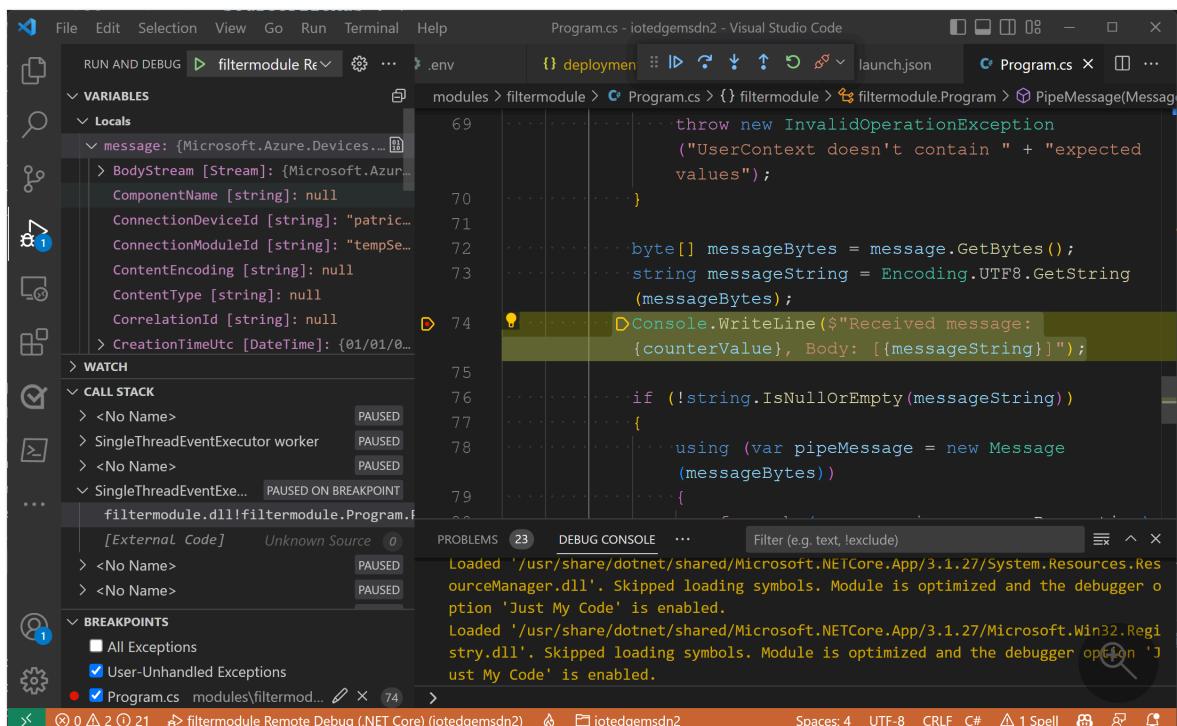
JSON

```
"configurations": [
{
    "name": "Remote Debug IoT Edge Module (.NET Core)",
    "type": "coreclr",
    "request": "attach",
    "processId": "${command:pickRemoteProcess}",
    "pipeTransport": {
        "pipeProgram": "docker",
        "pipeArgs": [
            "-H",
            "ssh://user@my-device-vm.eastus.cloudapp.azure.com:22",
            "exec",
            "-i",
            "filtermodule",
            "sh",
            "-c"
        ],
        "debuggerPath": "~/vsdbg/vsdbg",
        "pipeCwd": "${workspaceFolder}",
        "quoteArgs": true
    },
    "sourceFileMap": {
        "/app": "${workspaceFolder}/modules/filtermodule"
```

```
},
"justMyCode": true
},
```

Remotely debug your module

1. In Visual Studio Code Debug view, select the debug configuration *Remote Debug IoT Edge Module (.NET Core)*.
2. Select **Start Debugging** or select **F5**. Select the process to attach to.
3. In the Visual Studio Code Debug view, you see the variables in the left panel.
4. In Visual Studio Code, set breakpoints in your custom module.
5. When a breakpoint is hit, you can inspect variables, step through code, and debug your module.



ⓘ Note

The preceding example shows how to debug IoT Edge modules on remote containers. The example adds a remote Docker context and changes to the Docker privileges on the remote device. After you finish debugging your modules, set your Docker context to *default* and remove privileges from your user account.

See this [IoT Developer blog entry](#) for an example using a Raspberry Pi device.

Next steps

After you've built your module, learn how to [deploy Azure IoT Edge modules from Visual Studio Code](#).

To develop modules for your IoT Edge devices, understand and use [Azure IoT Hub SDKs](#).

Deploy Azure IoT Edge modules from the Azure portal

Article • 06/13/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Important

Starting August 28th 2024, Azure Marketplace is updating the distribution model for IoT Edge modules. Partners (module publishers) will begin [hosting their IoT Edge modules](#) ↗ on publisher-owned container registries. IoT Edge module images won't be available for download from the Azure Marketplace container registry.

Contact the [IoT Edge module publisher](#) ↗ to obtain the updated container image URI and [update your IoT Edge device configurations](#) with the new image URI provided by the publisher.

IoT Edge devices that don't use [partner modules](#) ↗ acquired from Azure Marketplace aren't affected and no action is required.

Once you create IoT Edge modules with your business logic, you want to deploy them to your devices to operate at the edge. If you have multiple modules that work together to collect and process data, you can deploy them all at once and declare the routing rules that connect them.

This article shows how the Azure portal guides you through creating a deployment manifest and pushing the deployment to an IoT Edge device. For information about creating a deployment that targets multiple devices based on their shared tags, see [Deploy and monitor IoT Edge modules at scale](#).

Prerequisites

- An [IoT Hub](#) in your Azure subscription.

- An IoT Edge device.

If you don't have an IoT Edge device set up, you can create one in an Azure virtual machine. Follow the steps in one of the quickstart articles to [Create a virtual Linux device](#) or [Create a virtual Windows device](#).

Configure a deployment manifest

A deployment manifest is a JSON document that describes which modules to deploy, how data flows between the modules, and desired properties of the module twins. For more information about how deployment manifests work and how to create them, see [Understand how IoT Edge modules can be used, configured, and reused](#).

The Azure portal has a wizard that walks you through creating the deployment manifest, instead of building the JSON document manually. It has three steps: **Add modules**, **Specify routes**, and **Review deployment**.

 **Note**

The steps in this article reflect the latest schema version of the IoT Edge agent and hub. Schema version 1.1 was released along with IoT Edge version 1.0.10, and enables the module startup order and route prioritization features.

If you are deploying to a device running version 1.0.9 or earlier, edit the **Runtime Settings** in the **Modules** step of the wizard to use schema version 1.0.

Select device and add modules

1. Sign in to the [Azure portal](#) and navigate to your IoT Hub.
2. On the left pane, select **Devices** under the **Device management** menu.
3. Select the target IoT Edge device from the list.
4. On the upper bar, select **Set Modules**.
5. In the **Container Registry Credentials** section of the page, provide credentials to access container registries that contain module images. For example, your modules are in your private container registry or you are using a partner container registry that requires authentication.
6. In the **IoT Edge Modules** section of the page, select **Add**.

The screenshot shows the Azure IoT Hub interface for managing device modules. At the top, it says "Set modules on device: mydevice". Below that, there's a "Container Registry Credentials" section where you can enter a name (myacr), address (myacr.azurecr.io), and user name (myusername). There's also a "Name" and "Address" field for a second entry. Under "IoT Edge Modules", there's a dropdown menu with "Add" and "Runtime Settings" options. It lists two modules: "IoT Edge Module" and "Azure Stream Analytics Module", both set to "running" status. A checkbox at the bottom allows sending usage data to Microsoft.

7. Choose the type of modules you want to add from the drop-down menu. You can add IoT Edge modules or Azure Stream Analytics modules.

IoT Edge Module

Use this option to add Microsoft modules, partner modules, or custom modules. You provide the module name and container image URI. The container image URI is the location of the module image in a container registry. For a list of Microsoft IoT Edge module images, see the [Microsoft Artifact Registry](#). For partner modules, contact the IoT Edge module publisher to obtain the container image URI.

For example to add the Microsoft simulated temperature sensor module:

1. Enter the following settings:

[Expand table](#)

Setting	Value
Image URI	mcr.microsoft.com/azureiotedge-simulated-temperature-sensor
Restart Policy	always
Desired Status	running

Add IoT Edge Module

iot-hub

IoT Edge module settings. [Learn more](#)

Module name *

SimulatedTemperatureSensor

Settings

Environment Variables

Container Create Options

Module Twin Settings

Image URI *

mcr.microsoft.com/azureiotedge-simulated-temperature-sensor

Restart Policy *

always

Desired Status *

running

Image Pull Policy

Always

Startup Order

200

2. Select Add.

3. After adding a module, select the module name from the list to open the module settings. Fill out the optional fields if necessary.

IoT Edge Modules

IoT Edge modules are Docker containers deployed to IoT Edge devices. They can communicate with other modules or send data to the IoT Edge runtime. Modules on a tier and units. For example, for S1 tier, modules can be set 10 times per second if no other updates are happening in the IoT Hub.

+ Add   Runtime Settings

NAME	DESIRED STATUS
SimulatedTemperatureSensor	running
PartnerModule	running

For more information about the available module settings, see [Module configuration and management](#).

For more information about the module twin, see [Define or update desired properties](#).

Azure Stream Analytics Module

Use this option for modules generated from an Azure Stream Analytics workload.

1. Select your subscription and the Azure Stream Analytics Edge job that you created.
2. Select Save.

For more information about deploying Azure Stream Analytics in an IoT Edge module, see [Tutorial: Deploy Azure Stream Analytics as an IoT Edge module](#).

Specify routes

On the **Routes** tab, you define how messages are passed between modules and the IoT Hub. Messages are constructed using name/value pairs. By default, the first deployment for a new device includes a route called **route** and defined as **FROM /messages/* INTO \$upstream**, which means that any messages output by any modules are sent to your IoT Hub.

The **Priority** and **Time to live** parameters are optional parameters that you can include in a route definition. The priority parameter allows you to choose which routes should have their messages processed first, or which routes should be processed last. Priority is determined by setting a number 0-9, where 0 is top priority. The time to live parameter allows you to declare how long messages in that route should be held until they're either processed or removed from the queue.

For more information about how to create routes, see [Declare routes](#).

Once the routes are set, select **Next: Review + create** to continue to the next step of the wizard.

Review deployment

The review section shows you the JSON deployment manifest that was created based on your selections in the previous two sections. Note that there are two modules declared that you didn't add: **\$edgeAgent** and **\$edgeHub**. These two modules make up the **IoT Edge runtime** and are required defaults in every deployment.

Review your deployment information, then select **Create**.

View modules on your device

Once you've deployed modules to your device, you can view all of them in the device details page of your IoT Hub. This page displays the name of each deployed module, as well as useful information like the deployment status and exit code.

Select **Next: Routes** and continue with deployment as described by [Specify routes](#) and [Review Deployment](#) earlier in this article.

Next steps

Learn how to [Deploy and monitor IoT Edge modules at scale](#).

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback ↗

Deploy Azure IoT Edge modules with Azure CLI

Article • 05/01/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Once you create Azure IoT Edge modules with your business logic, you want to deploy them to your devices to operate at the edge. If you have multiple modules that work together to collect and process data, you can deploy them all at once. You can also declare the routing rules that connect them.

[Azure CLI](#) is an open-source cross platform, command-line tool for managing Azure resources such as IoT Edge. It enables you to manage Azure IoT Hub resources, device provisioning service instances, and linked-hubs out of the box. The new IoT extension enriches Azure CLI with features such as device management and full IoT Edge capability.

This article shows how to create a JSON deployment manifest, then use that file to push the deployment to an IoT Edge device. For information about creating a deployment that targets multiple devices based on their shared tags, see [Deploy and monitor IoT Edge modules at scale](#)

Prerequisites

- An [IoT hub](#) in your Azure subscription.
- An IoT Edge device

If you don't have an IoT Edge device set up, you can create one in an Azure virtual machine. Follow the steps in one of the quickstart articles to [Create a virtual Linux device](#) or [Create a virtual Windows device](#).

- [Azure CLI](#) in your environment. At a minimum, your Azure CLI version must be 2.0.70 or higher. Use `az --version` to validate. This version supports az extension

commands and introduces the Knack command framework.

- The [IoT extension for Azure CLI](#).

Configure a deployment manifest

A deployment manifest is a JSON document that describes which modules to deploy, how data flows between the modules, and desired properties of the module twins. For more information about how deployment manifests work and how to create them, see [Understand how IoT Edge modules can be used, configured, and reused](#).

To deploy modules using the Azure CLI, save the deployment manifest locally as a .json file. You use the file path in the next section when you run the command to apply the configuration to your device.

Here's a basic deployment manifest with one module as an example:

ⓘ Note

This sample deployment manifest uses schema version 1.1 for the IoT Edge agent and hub. Schema version 1.1 was released along with IoT Edge version 1.0.10, and enables features like module startup order and route prioritization.

JSON

```
{  
  "content": {  
    "modulesContent": {  
      "$edgeAgent": {  
        "properties.desired": {  
          "schemaVersion": "1.1",  
          "runtime": {  
            "type": "docker",  
            "settings": {  
              "minDockerVersion": "v1.25",  
              "loggingOptions": "",  
              "registryCredentials": {}  
            }  
          },  
          "systemModules": {  
            "edgeAgent": {  
              "type": "docker",  
              "settings": {  
                "image": "mcr.microsoft.com/azureiotedge-agent:1.5",  
                "createOptions": "{}"  
              }  
            },  
            "edgeHub": {  
              "type": "edgeHub",  
              "settings": {}  
            }  
          }  
        }  
      }  
    }  
  }  
}
```

```
"edgeHub": {
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
        "image": "mcr.microsoft.com/azureiotedge-hub:1.5",
        "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}\"}"
    }
},
"modules": {
    "SimulatedTemperatureSensor": {
        "version": "1.0",
        "type": "docker",
        "status": "running",
        "restartPolicy": "always",
        "settings": {
            "image": "mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.5",
            "createOptions": "{}"
        }
    }
},
"$edgeHub": {
    "properties.desired": {
        "schemaVersion": "1.1",
        "routes": {
            "upstream": "FROM /messages/* INTO $upstream"
        },
        "storeAndForwardConfiguration": {
            "timeToLiveSecs": 7200
        }
    }
},
"SimulatedTemperatureSensor": {
    "properties.desired": {
        "SendData": true,
        "SendInterval": 5
    }
}
```

Deploy to your device

You deploy modules to your device by applying the deployment manifest that you configured with the module information.

Change directories into the folder where you saved your deployment manifest. If you used one of the Visual Studio Code IoT Edge templates, use the `deployment.json` file in the **config** folder of your solution directory and not the `deployment.template.json` file.

Use the following command to apply the configuration to an IoT Edge device:

Azure CLI

```
az iot edge set-modules --device-id [device id] --hub-name [hub name] --content [file path]
```

The device ID parameter is case-sensitive. The content parameter points to the deployment manifest file that you saved.

```
kg@IoTUbuntu:~/edgeSolution/config$ az iot edge set-modules -d edge001 -n CLIDemoHub -k ./deployment.json
[
  {
    "authenticationType": "sas",
    "cloudToDeviceMessageCount": 0,
    "connectionState": "Connected",
    "deviceetag": "NzQ3NjY1NjYz",
    "deviceId": "edge001",
    "etag": "AAAAAAAAAAI=",
    "lastActivityTime": "0001-01-01T00:00:00+00:00",
    "moduleId": "$edgeAgent",
    "properties": {
      "desired": {
        "$metadata": {
          "$lastUpdated": "2018-07-27T18:03:24.6210811Z",
          "$lastUpdatedVersion": 2,
          "modules": {
            "$lastUpdated": "2018-07-27T18:03:24.6210811Z",
            "$lastUpdatedVersion": 2,
            "tempSensor": {
              "$lastUpdated": "2018-07-27T18:03:24.6210811Z",
              "$lastUpdatedVersion": 2,
              "restartPolicy": {
                "$lastUpdated": "2018-07-27T18:03:24.6210811Z",
                "$lastUpdatedVersion": 2
              },
              "settings": {
                "$lastUpdated": "2018-07-27T18:03:24.6210811Z",
                "$lastUpdatedVersion": 2,
                "createOptions": {

```

View modules on your device

Once you've deployed modules to your device, you can view all of them with the following command:

View the modules on your IoT Edge device:

Azure CLI

```
az iot hub module-identity list --device-id [device id] --hub-name [hub name]
```

The device ID parameter is case-sensitive.

```
kg@IoTUbuntu:~$ az iot hub module-identity list --device-id edge001 --hub-name CLIDemoHub
[
  {
    "authenticationType": "none",
    "cloudToDeviceMessageCount": 0,
    "connectionState": "Disconnected",
    "deviceId": "edge001",
    "etag": "AAAAAAAAAAI=",
    "lastActivityTime": "0001-01-01T00:00:00",
    "moduleId": "$edgeAgent",
    "properties": {
      "desired": {
        "$metadata": [
          {"$lastUpdated": "2018-02-12T21:07:01.191736Z",
           "$lastUpdatedVersion": 2},
          {"modules": [
            {"$lastUpdated": "2018-02-12T21:07:01.191736Z",
             "$lastUpdatedVersion": 2,
             "tempSensor": {
               "$lastUpdated": "2018-02-12T21:07:01.191736Z",
               "$lastUpdatedVersion": 2,
               "restartPolicy": {
                 "$lastUpdated": "2018-02-12T21:07:01.191736Z",
                 "$lastUpdatedVersion": 2
               },
               "settings": [
                 {"$lastUpdated": "2018-02-12T21:07:01.191736Z",
                  "$lastUpdatedVersion": 2,
                  "createOptions": [
                    {"$lastUpdated": "2018-02-12T21:07:01.191736Z",
                     "$lastUpdatedVersion": 2
                   }
                 ],
                 "image": {
                   "$lastUpdated": "2018-02-12T21:07:01.191736Z"
                 }
               }
             }
           ]
         }
       }
     }
   }
]
```

Next steps

Learn how to [Deploy and monitor IoT Edge modules at scale](#)

Deploy IoT Edge modules at scale using the Azure portal

Article • 06/12/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Create an **IoT Edge automatic deployment** in the Azure portal to manage ongoing deployments for many devices at once. Automatic deployments for IoT Edge are part of the [device management](#) feature of IoT Hub. Deployments are dynamic processes that enable you to deploy multiple modules to multiple devices, track the status and health of the modules, and make changes when necessary.

For more information, see [Understand IoT Edge automatic deployments for single devices or at scale](#).

Identify devices using tags

Before you can create a deployment, you have to be able to specify which devices you want to affect. Azure IoT Edge identifies devices using **tags** in the device twin. Each device can have multiple tags that you define in any way that makes sense for your solution.

For example, if you manage a campus of smart buildings, you might add location, room type, and environment tags to a device:

JSON

```
"tags":{  
    "location":{  
        "building": "20",  
        "floor": "2"  
    },  
    "roomtype": "conference",  
    "environment": "prod"  
}
```

For more information about device twins and tags, see [Understand and use device twins in IoT Hub](#).

Create a deployment

IoT Edge provides two different types of automatic deployments that you can use to customize your scenario. You can create a standard *deployment*, which includes that system runtime modules and any additional modules and routes. Each device can only apply one deployment. Or you can create a *layered deployment*, which only includes custom modules and routes, not the system runtime. Many layered deployments can be combined on a device, on top of a standard deployment. For more information about how the two types of automatic deployments work together, see [Understand IoT Edge automatic deployments for single devices or at scale](#).

The steps for creating a deployment and a layered deployment are similar. Any differences are called out in the following steps.

1. In the [Azure portal](#), go to your IoT Hub.
2. On the menu in the left pane, select **Configurations + Deployments** under **Device Management**.
3. On the upper bar, select **Add > Add Deployment** or **Add Layered Deployment**.

There are five steps to create a deployment. The following sections walk through each one.

ⓘ Note

The steps in this article reflect the latest schema version of the IoT Edge agent and hub.

If you are deploying to a device running version 1.0.9 or earlier, edit the **Runtime Settings** in the **Modules** step of the wizard to use schema version 1.0.

Step 1: Name and label

1. Give your deployment a unique name that is up to 128 lowercase letters. Avoid spaces and the following invalid characters: & ^ [] { } \ | " < > /.
2. You can add labels as key-value pairs to help track your deployments. For example, **HostPlatform** and **Linux**, or **Version** and **3.0.1**.
3. Select **Next: Modules** to move to step two.

Step 2: Modules

You can add up to 50 modules to a deployment. If you create a deployment with no modules, it removes any current modules from the target devices.

In deployments, you can manage the settings for the IoT Edge agent and IoT Edge hub modules. Select **Runtime Settings** to configure the two runtime modules. In layered deployment, the runtime modules are not included so cannot be configured.

To add custom code as a module, or to manually add an Azure service module, follow these steps:

1. In the **Container Registry Settings** section of the page, provide the credentials to access any private container registries that contain your module images.
2. In the **IoT Edge Modules** section of the page, select **Add**.
3. Choose one of the types of modules from the drop-down menu:
 - **IoT Edge Module** - You provide the module name and container image URI. For example, the image URI for the sample SimulatedTemperatureSensor module is `mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.0`. For a list of Microsoft IoT Edge module images, see the [Microsoft Artifact Registry](#).
 - **Azure Stream Analytics Module** - Modules generated from an Azure Stream Analytics workload.
4. If needed, repeat steps 2 and 3 to add additional modules to your deployment.

After you add a module to a deployment, you can select its name to open the **Update IoT Edge Module** page. On this page, you can edit the module settings, environment variables, create options, startup order, and module twin. If you added a module from the marketplace, it may already have some of these parameters filled in. For more information about the available module settings, see [Module configuration and management](#).

If you're creating a layered deployment, you may be configuring a module that exists in other deployments targeting the same devices. To update the module twin without overwriting other versions, open the **Module Twin Settings** tab. Create a new **Module Twin Property** with a unique name for a subsection within the module twin's desired properties, for example `properties.desired.settings`. If you define properties within just the `properties.desired` field, it overwrites the desired properties for the module defined in any lower priority deployments.

Add IoT Edge Module

Specify the settings for an IoT Edge custom module.
[Learn more](#)

IoT Edge Module Name *

SimulatedTemperatureSensor

Module Settings Environment Variables Container Create Options **Module Twin Settings**

The IoT Hub will update sections of the IoT Edge module twin's desired properties to reflect listed entries.
[Learn more about deployments](#)

Module Twin Property

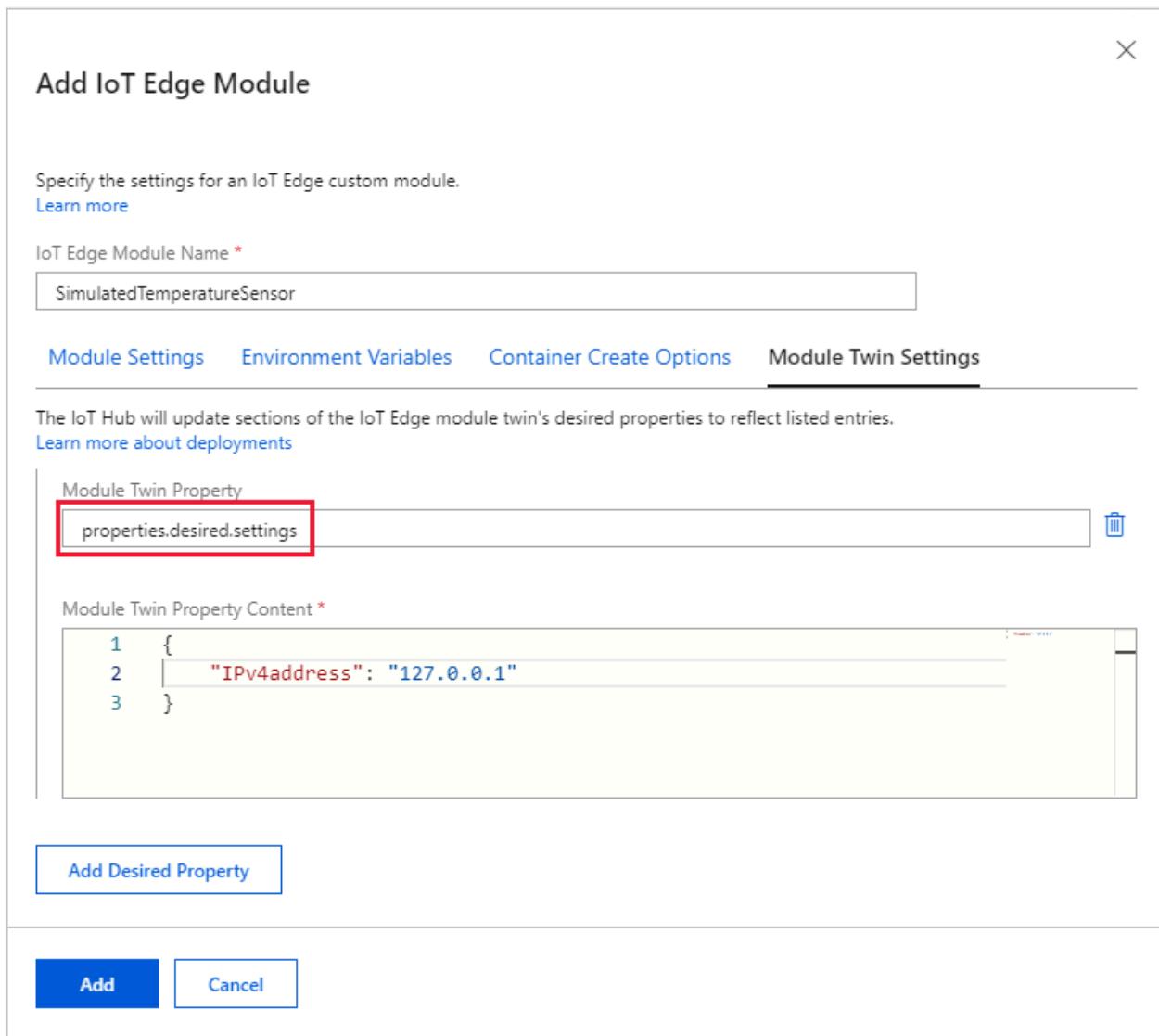
properties.desired.settings

Module Twin Property Content *

```
1 {
2   "IPv4address": "127.0.0.1"
3 }
```

Add Desired Property

Add Cancel



For more information about module twin configuration in layered deployments, see [Layered deployment](#).

Once you have all the modules for a deployment configured, select **Next: Routes** to move to step three.

Step 3: Routes

On the **Routes** tab, you define how messages are passed between modules and the IoT Hub. Messages are constructed using name/value pairs.

For example, a route with a name **route** and a value **FROM /messages/* INTO \$upstream** would take any messages output by any modules and send them to your IoT hub.

The **Priority** and **Time to live** parameters are optional parameters that you can include in a route definition. The priority parameter allows you to choose which routes should have their messages processed first, or which routes should be processed last. Priority is determined by setting a number 0-9, where 0 is top priority. The time to live parameter

allows you to declare how long messages in that route should be held until they're either processed or removed from the queue.

For more information about how to create routes, see [Declare routes](#).

Select **Next: Target Devices**.

Step 4: Target devices

Use the `tags` property from your devices to target the specific devices that should receive this deployment.

Since multiple deployments may target the same device, you should give each deployment a priority number. If there's ever a conflict, the deployment with the highest priority (larger values indicate higher priority) wins. If two deployments have the same priority number, the one that was created most recently wins.

If multiple deployments target the same device, then only the one with the higher priority is applied. If multiple layered deployments target the same device then they are all applied. However, if any properties are duplicated, like if there are two routes with the same name, then the one from the higher priority layered deployment overwrites the rest.

Any layered deployment targeting a device must have a higher priority than the base deployment in order to be applied.

1. Enter a positive integer for the deployment **Priority**.
2. Enter a **Target condition** to determine which devices are targeted with this deployment. The condition is based on device twin tags or device twin reported properties and should match the expression format. For example,
`tags.environment='test' or properties.reported.devicemodel='4000x'`.

Select **Next: Metrics**.

Step 5: Metrics

Metrics provide summary counts of the various states that a device may report back as a result of applying configuration content.

1. Enter a name for **Metric Name**.
2. Enter a query for **Metric Criteria**. The query is based on IoT Edge hub module twin [reported properties](#). The metric represents the number of rows returned by the query.

For example:

SQL

```
SELECT deviceId FROM devices  
WHERE properties.reported.lastDesiredStatus.code = 200
```

Select **Next: Review + Create** to move on to the final step.

Step 6: Review and create

Review your deployment information, then select **Create**.

To monitor your deployment, see [Monitor IoT Edge deployments](#).

ⓘ Note

When a new IoT Edge deployment is created, sometimes it can take up to 5 minutes for the IoT Hub to process the new configuration and propagate the new desired properties to the targeted devices.

Modify a deployment

When you modify a deployment, the changes immediately replicate to all targeted devices. You can modify the following settings and features for an existing deployment:

- Target conditions
- Custom metrics
- Labels
- Tags
- Desired properties

Modify target conditions, custom metrics, and labels

1. In your IoT hub, select **Configurations + Deployments** from the left pane menu.
2. Select the deployment you want to configure.
3. Select the **Target Devices** tab. Change the **Target Condition** to target the intended devices. You can also adjust the **Priority**.

If you update the target condition, the following updates occur:

- If a device didn't meet the old target condition, but meets the new target condition and this deployment is the highest priority for that device, then this deployment is applied to the device.
- If a device currently running this deployment no longer meets the target condition, it uninstalls this deployment and takes on the next highest priority deployment.
- If a device currently running this deployment no longer meets the target condition and doesn't meet the target condition of any other deployments, then no change occurs on the device. The device continues running its current modules in their current state, but is not managed as part of this deployment anymore. Once it meets the target condition of any other deployment, it uninstalls this deployment and takes on the new one.

4. Select the **Metrics** tab and select the **Edit Metrics** button. Add or modify custom metrics, using the example syntax as a guide. Select **Save**.

The screenshot shows the 'Deployment Details' page for a deployment named 'deploy-main'. At the top, there are buttons for Save, Clone, Download Deployment File, and Download Deployment Manifest. Below these are two input fields: '10' and '10'. Underneath are tabs for Target Condition, Metrics (which is selected), Labels, Modules, Routes, and Deployment. A tooltip for the Metrics tab states: 'Metrics provide an aggregate view of devices scoped by a deployment. These totals (e.g. total targeted devices or total devices with applied deployment settings) inform the overall state of the deployment.' The main area displays a table of metrics:

Metric Name	Metric Criteria
Targeted	select deviceld from devices where capabilities.iotEdge = true and tags.environment = 'Production'
Applied	select deviceld from devices.modules where moduleId = '\$edgeAgent' and connectionStatus = 'Success'
Reporting Success	select deviceld from devices.modules where moduleId = '\$edgeAgent' and connectionStatus = 'Success'
Reporting Failure	select deviceld from devices.modules where moduleId = '\$edgeAgent' and connectionStatus = 'Failure'
e.g. successfullyConfigured	e.g. SELECT deviceld FROM devices WHERE properties.reported.lastDesiredStatus = 'Success'

At the bottom left is a 'View Results' button, and at the bottom right is a trash can icon for deleting the deployment.

5. Select the **Labels** tab and make any desired changes and select **Save**.

Delete a deployment

When you delete a deployment, any deployed devices take on their next highest priority deployment. If your devices don't meet the target condition of any other deployment, then the modules are not removed when the deployment is deleted.

1. Sign in to the [Azure portal](#) and navigate to your IoT Hub.
2. Select **Configurations + Deployments**.
3. Use the checkbox to select the deployment that you want to delete.
4. Select **Delete**.

5. A prompt informs you that this action deletes this deployment and revert to the previous state for all devices. A deployment with a lower priority will apply. If no other deployment is targeted, no modules are removed. If you want to remove all modules from your device, create a deployment with zero modules and deploy it to the same devices. Select **Yes** to continue.

Next steps

Learn more about [Deploying modules to IoT Edge devices](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Deploy and monitor IoT Edge modules at scale by using the Azure CLI

Article • 03/22/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Create an [Azure IoT Edge automatic deployment](#) by using the Azure CLI to manage ongoing deployments for many devices at once. Automatic deployments for IoT Edge are part of the [device management](#) feature of Azure IoT Hub. Deployments are dynamic processes that enable you to deploy multiple modules to multiple devices, track the status and health of the modules, and make changes when necessary.

In this article, you set up the Azure CLI and the IoT extension. You then learn how to deploy modules to a set of IoT Edge devices and monitor the progress by using the available CLI commands.

Prerequisites

- An [IoT hub](#) in your Azure subscription.
- One or more IoT Edge devices.

If you don't have an IoT Edge device set up, you can create one in an Azure virtual machine. Follow the steps in one of these quickstart articles: [Create a virtual Linux device](#) or [Create a virtual Windows device](#).

- The [Azure CLI](#) in your environment. Your Azure CLI version must be 2.0.70 or later. Use `az --version` to validate. This version supports az extension commands and introduces the Knack command framework.
- The [IoT extension for Azure CLI](#).

Configure a deployment manifest

A deployment manifest is a JSON document that describes which modules to deploy, how data flows between the modules, and desired properties of the module twins. For more information, see [Learn how to deploy modules and establish routes in IoT Edge](#).

To deploy modules by using the Azure CLI, save the deployment manifest locally as a .txt file. You use the file path in the next section when you run the command to apply the configuration to your device.

Here's a basic deployment manifest with one module as an example:

```
JSON

{
  "content": {
    "modulesContent": {
      "$edgeAgent": {
        "properties.desired": {
          "schemaVersion": "1.1",
          "runtime": {
            "type": "docker",
            "settings": {
              "minDockerVersion": "v1.25",
              "loggingOptions": "",
              "registryCredentials": {}
            }
          }
        },
        "systemModules": {
          "edgeAgent": {
            "type": "docker",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-agent:1.5",
              "createOptions": "{}"
            }
          },
          "edgeHub": {
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-hub:1.5",
              "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}}}"
            }
          }
        },
        "modules": {
          "SimulatedTemperatureSensor": {
            "version": "1.5",
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-simulation:1.5",
              "createOptions": "{}"
            }
          }
        }
      }
    }
  }
}
```

```
        "settings": {
            "image": "mcr.microsoft.com/azureiotedge-simulated-
temperature-sensor:1.5",
            "createOptions": "{}"
        }
    }
}
},
"$edgeHub": {
    "properties.desired": {
        "schemaVersion": "1.1",
        "routes": {
            "upstream": "FROM /messages/* INTO $upstream"
        },
        "storeAndForwardConfiguration": {
            "timeToLiveSecs": 7200
        }
    }
},
"SimulatedTemperatureSensor": {
    "properties.desired": {
        "SendData": true,
        "SendInterval": 5
    }
}
}
}
```

ⓘ Note

This sample deployment manifest uses schema version 1.1 for the IoT Edge agent and hub. Schema version 1.1 was released along with IoT Edge version 1.0.10. It enables features like module startup order and route prioritization.

Layered deployment

Layered deployments are a type of automatic deployment that can be stacked on top of each other. For more information about layered deployments, see [Understand IoT Edge automatic deployments for single devices or at scale](#).

Layered deployments can be created and managed with the Azure CLI like any automatic deployment, with just a few differences. After a layered deployment is created, the Azure CLI works for layered deployments the same as it does for any deployment. To create a layered deployment, add the `--layered` flag to the create command.

The second difference is in the construction of the deployment manifest. Whereas standard automatic deployment must contain the system runtime modules in addition to any user modules, layered deployments can contain only user modules. Layered deployments also need a standard automatic deployment on a device to supply the required components of every IoT Edge device, like the system runtime modules.

Here's a basic layered deployment manifest with one module as an example:

```
JSON

{
  "content": {
    "modulesContent": {
      "$edgeAgent": {
        "properties.desired.modules.SimulatedTemperatureSensor": {
          "settings": {
            "image": "mcr.microsoft.com/azureiotedge-simulated-temperature-
sensor:1.5",
            "createOptions": "{}"
          },
          "type": "docker",
          "status": "running",
          "restartPolicy": "always",
          "version": "1.5"
        }
      },
      "$edgeHub": {
        "properties.desired.routes.upstream": "FROM /messages/* INTO
$upstream"
      },
      "SimulatedTemperatureSensor": {
        "properties.desired": {
          "SendData": true,
          "SendInterval": 5
        }
      }
    }
  }
}
```

ⓘ Note

This layered deployment manifest has a slightly different format from a standard deployment manifest. The desired properties of the runtime modules are collapsed and use dot notation. This formatting is required for the Azure portal to recognize a layered deployment. For example:

- `properties.desired.modules.<module_name>`

- `properties.desired.routes.<route_name>`

The previous example showed the layered deployment setting `properties.desired` for a module. If this layered deployment targeted a device where the same module was already applied, it would overwrite any existing desired properties. To update desired properties instead of overwriting them, you can define a new subsection. For example:

JSON

```
"SimulatedTemperatureSensor": {
  "properties.desired.layeredProperties": {
    "SendData": true,
    "SendInterval": 5
  }
}
```

The same can also be expressed with:

JSON

```
"SimulatedTemperatureSensor": {
  "properties.desired.layeredProperties.SendData" : true,
  "properties.desired.layeredProperties.SendInterval": 5
}
```

① Note

Currently, all layered deployments must include an `edgeAgent` object to be considered valid. Even if a layered deployment only updates module properties, include an empty object. For example: `["$edgeAgent":{}]`. A layered deployment with an empty `edgeAgent` object will be shown as **targeted** in the `edgeAgent` module twin, not **applied**.

In summary, to create a layered deployment:

- Add the `--layered` flag to the Azure CLI create command.
- Don't include system modules.
- Use full dot notation under `$edgeAgent` and `$edgeHub`.

For more information about configuring module twins in layered deployments, see [Layered deployment](#).

Identify devices by using tags

Before you can create a deployment, you have to be able to specify which devices you want to affect. Azure IoT Edge identifies devices by using *tags* in the device twin.

Each device can have multiple tags that you define in any way that makes sense for your solution. For example, if you manage a campus of smart buildings, you might add the following tags to a device:

JSON

```
"tags":{  
    "location":{  
        "building": "20",  
        "floor": "2"  
    },  
    "roomtype": "conference",  
    "environment": "prod"  
}
```

For more information about device twins and tags, see [Understand and use device twins in IoT Hub](#).

Create a deployment

You deploy modules to your target devices by creating a deployment that consists of the deployment manifest and other parameters.

Use the [az iot edge deployment create](#) command to create a deployment:

Azure CLI

```
az iot edge deployment create --deployment-id [deployment id] --hub-name  
[hub name] --content [file path] --labels "[labels]" --target-condition "  
[target query]" --priority [int]
```

Use the same command with the `--layered` flag to create a layered deployment.

The create command for deployment takes the following parameters:

- **--layered**. An optional flag to identify the deployment as a layered deployment.
- **--deployment-id**. The name of the deployment that will be created in the IoT hub. Give your deployment a unique name that's up to 128 lowercase letters. Avoid spaces and the following invalid characters: & ^ [] { } \ | " < > /. This parameter is required.

- **--content**. File path to the deployment manifest JSON. This parameter is required.
- **--hub-name**. Name of the IoT hub in which the deployment will be created. The hub must be in the current subscription. Change your current subscription by using the `az account set -s [subscription name]` command.
- **--labels**. Name/value pairs that describe and help you track your deployments. Labels take JSON formatting for the names and values. For example:
`{"HostPlatform": "Linux", "Version": "3.0.1"}`.
- **--target-condition**. The condition that determines which devices will be targeted with this deployment. The condition is based on device twin tags or device twin reported properties, and it should match the expression format. For example:
`tags.environment='test' and properties.reported.devicemodel='4000x'`. If the target condition is not specified, then the deployment is not applied to any devices.
- **--priority**. A positive integer. If two or more deployments are targeted at the same device, the deployment with the highest numerical value for priority will apply.
- **--metrics**. Metrics that query the `edgeHub` reported properties to track the status of a deployment. Metrics take JSON input or a file path. For example:
`'{"queries": {"mymetric": "SELECT deviceId FROM devices WHERE properties.reported.lastDesiredStatus.code = 200"}}'`.

To monitor a deployment by using the Azure CLI, see [Monitor IoT Edge deployments](#).

Note

When a new IoT Edge deployment is created, sometimes it can take up to 5 minutes for the IoT Hub to process the new configuration and propagate the new desired properties to the targeted devices.

Modify a deployment

When you modify a deployment, the changes immediately replicate to all targeted devices.

If you update the target condition, the following updates occur:

- If a device didn't meet the old target condition but meets the new target condition, and this deployment is the highest priority for that device, then this deployment is applied to the device.
- If a device currently running this deployment no longer meets the target condition, it uninstalls this deployment and takes on the next-highest-priority deployment.

- If a device currently running this deployment no longer meets the target condition and doesn't meet the target condition of any other deployments, then no change occurs on the device. The device continues running its current modules in their current state but is not managed as part of this deployment anymore. After the device meets the target condition of any other deployment, it uninstalls this deployment and takes on the new one.

You can't update the content of a deployment, which includes the modules and routes defined in the deployment manifest. If you want to update the content of a deployment, create a new deployment that targets the same devices with a higher priority. You can modify certain properties of an existing module, including the target condition, labels, metrics, and priority.

Use the [az iot edge deployment update](#) command to update a deployment:

Azure CLI

```
az iot edge deployment update --deployment-id [deployment id] --hub-name
[hub name] --set [property1.property2='value']
```

The deployment update command takes the following parameters:

- **--deployment-id**. The name of the deployment that exists in the IoT hub.
- **--hub-name**. The name of the IoT hub in which the deployment exists. The hub must be in the current subscription. Switch to the desired subscription by using the command `az account set -s [subscription name]`.
- **--set**. Update a property in the deployment. You can update the following properties:
 - `targetCondition` (for example, `targetCondition=tags.location.state='Oregon'`)
 - `labels`
 - `priority`
- **--add**. Add a new property to the deployment, including target conditions or labels.
- **--remove**. Remove an existing property, including target conditions or labels.

Delete a deployment

When you delete a deployment, any devices take on their next-highest-priority deployment. If your devices don't meet the target condition of any other deployment, the modules are not removed when the deployment is deleted.

Use the [az iot edge deployment delete](#) command to delete a deployment:

Azure CLI

```
az iot edge deployment delete --deployment-id [deployment id] --hub-name  
[hub name]
```

The `deployment delete` command takes the following parameters:

- **--deployment-id**. The name of the deployment that exists in the IoT hub.
- **--hub-name**. The name of the IoT hub in which the deployment exists. The hub must be in the current subscription. Switch to the desired subscription by using the command `az account set -s [subscription name]`.

Next steps

Learn more about [deploying modules to IoT Edge devices](#).

Monitor IoT Edge deployments

Article • 06/03/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge provides reporting that lets you monitor real-time information on the modules deployed to your IoT Edge devices. The IoT Hub service retrieves the status from the devices and makes them available to the operator. Monitoring is also important for [deployments made at scale](#) that include automatic deployments and layered deployments.

Both devices and modules have similar data, such as connectivity, so values are obtained according to the device ID or the module ID.

The IoT Hub service collects data reported by device and module twins and provides counts of the various states that devices may have. The IoT Hub service organizes this data into four groups of metrics:

 [Expand table](#)

Type	Description
Targeted	Shows the IoT Edge devices that match the deployment targeting condition.
Applied	Shows the targeted IoT Edge devices that are not targeted by another deployment of higher priority.
Reporting Success	Shows the IoT Edge devices that have reported that the modules have been deployed successfully.
Reporting Failure	Shows the IoT Edge devices that have reported that one or more modules haven't been deployed successfully. To further investigate the error, connect remotely to those devices and view the log files.

The IoT Hub service makes this data available for you to monitor in the Azure portal and in the Azure CLI.

Monitor a deployment in the Azure portal

To view the details of a deployment and monitor the devices running it, use the following steps:

1. Sign in to the [Azure portal](#) and navigate to your IoT Hub.
2. Select **Configurations + Deployments** under the **Device management** menu.
3. Inspect the deployment list. For each deployment, you can view the following details:

 Expand table

Column	Description
ID	The name of the deployment.
Type	The type of deployment, either Deployment or Layered Deployment .
Target Condition	The tag used to define targeted devices.
Priority	The priority number assigned to the deployment.
System metrics	The number of device twins in IoT Hub that match the targeting condition. Applied specifies the number of devices that have had the deployment content applied to their module twins in IoT Hub.
Device Metrics	The number of IoT Edge devices reporting success or errors from the IoT Edge client runtime.
Custom Metrics	The number of IoT Edge devices reporting data for any metrics that you defined for the deployment.
Created	The timestamp from when the deployment was created. This timestamp is used to break ties when two deployments have the same priority.

4. Select the deployment that you want to monitor.
5. On the **Deployment Details** page, scroll down to the bottom section and select the **Target Condition** tab. Select **View** to list the devices that match the target condition. You can change the condition and also the **Priority**. Select **Save** if you made changes.

Home > corplioTHub | IoT Edge > Deployment Details

Deployment Details

Type Deployment

Priority * ⓘ 10

Target Condition Metrics Labels Modules Routes Deployment

The target condition defines the devices targeted by this deployment. The condition is evaluated continuously to detect devices that enter or exit the set of targeted devices.

Target Condition ⓘ
tags.environment='test'

View

6. Select the **Metrics** tab. If you choose a metric from the **Select Metric** drop-down, a **View** button appears for you to display the results. You can also select **Edit Metrics** to adjust the criteria for any custom metrics that you have defined. Select **Save** if you made changes.

Home > IoTEdgeAndMLHub-jrujej6de6i7w | IoT Edge > Deployment Details

Deployment Details

Type Deployment

Sat Apr 11 2020 21:55:28 GMT-0700 (Pacific Daylight Time)

Priority * ⓘ 10

Target Condition **Metrics** Labels Modules Routes Deployment

Metrics provide an aggregate view of devices scoped by a deployment. These totals (e.g. total targeted devices or total devices with applied deployment settings) inform the overall state of the deployment.

Select Metric ⓘ
select metric

Edit Metrics

To make changes to your deployment, see [Modify a deployment](#).

Monitor a deployment with Azure CLI

Use the `az iot edge deployment show` command to display the details of a single deployment:

Azure CLI

```
az iot edge deployment show --deployment-id [deployment id] --hub-name [hub name]
```

The deployment show command takes the following parameters:

- **--deployment-id** - The name of the deployment that exists in the IoT hub.
Required parameter.
- **--hub-name** - Name of the IoT hub in which the deployment exists. The hub must be in the current subscription. Switch to the desired subscription with the command `az account set -s [subscription name]`

Inspect the deployment in the command window. The **metrics** property lists a count for each metric that is evaluated by each hub:

- **targetedCount** - A system metric that specifies the number of device twins in IoT Hub that match the targeting condition.
- **appliedCount** - A system metric specifies the number of devices that have had the deployment content applied to their module twins in IoT Hub.
- **reportedSuccessfulCount** - A device metric that specifies the number of IoT Edge devices in the deployment reporting success from the IoT Edge client runtime.
- **reportedFailedCount** - A device metric that specifies the number of IoT Edge devices in the deployment reporting failure from the IoT Edge client runtime.

You can show a list of device IDs or objects for each of the metrics with the [az iot edge deployment show-metric](#) command:

Azure CLI

```
az iot edge deployment show-metric --deployment-id [deployment id] --metric-id [metric id] --hub-name [hub name]
```

The deployment show-metric command takes the following parameters:

- **--deployment-id** - The name of the deployment that exists in the IoT hub.
- **--metric-id** - The name of the metric for which you want to see the list of device IDs, for example `reportedFailedCount`.
- **--hub-name** - Name of the IoT hub in which the deployment exists. The hub must be in the current subscription. Switch to the desired subscription with the command `az account set -s [subscription name]`.

To make changes to your deployment, see [Modify a deployment](#).

Next steps

Learn how to [monitor module twins](#), primarily the IoT Edge Agent and IoT Edge Hub runtime modules, for the connectivity and health of your IoT Edge deployments.

Monitor module twins

Article • 06/03/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Module twins in Azure IoT Hub enable monitoring the connectivity and health of your IoT Edge deployments. Module twins store useful information in your IoT hub about the performance of your running modules. The [IoT Edge agent](#) and the [IoT Edge hub](#) runtime modules each maintain their module twins, `$edgeAgent` and `$edgeHub`, respectively:

- `$edgeAgent` contains health and connectivity data about both the IoT Edge agent and IoT Edge hub runtime modules, and your custom modules. The IoT Edge agent is responsible for deploying the modules, monitoring them, and reporting connection status to your Azure IoT hub.
- `$edgeHub` contains data about communications between the IoT Edge hub running on a device and your Azure IoT hub. This includes processing incoming messages from downstream devices. IoT Edge hub is responsible for processing the communications between the Azure IoT Hub and the IoT Edge devices and modules.

The data is organized into metadata, tags, along with desired and reported property sets in the module twins' JSON structures. The desired properties you specified in your deployment.json file are copied to the module twins. The IoT Edge agent and the IoT Edge hub each update the reported properties for their modules.

Similarly, the desired properties specified for your custom modules in the deployment.json file are copied to its module twin, but your solution is responsible for providing its reported property values.

This article describes how to review the module twins in the Azure portal, Azure CLI, and in Visual Studio Code. For information on monitoring how your devices receive the deployments, see [Monitor IoT Edge deployments](#). For an overview on the concept of module twins, see [Understand and use module twins in IoT Hub](#).

💡 Tip

The reported properties of a runtime module could be stale if an IoT Edge device gets disconnected from its IoT hub. You can [ping](#) the `$edgeAgent` module to determine if the connection was lost.

Monitor runtime module twins

To troubleshoot deployment connectivity issues, review the IoT Edge agent and IoT Edge hub runtime module twins and then drill down into other modules.

Monitor IoT Edge agent module twin

The following JSON shows the `$edgeAgent` module twin in Visual Studio Code with most of the JSON sections collapsed.

JSON

```
{  
    "deviceId": "Windows109",  
    "moduleId": "$edgeAgent",  
    "etag": "AAAAAAAAAAU=",  
    "deviceEtag": "NzgwNjA1MDUz",  
    "status": "enabled",  
    "statusUpdateTime": "2001-01-01T00:00:00Z",  
    "connectionState": "Disconnected",  
    "lastActivityTime": "2001-01-01T00:00:00Z",  
    "cloudToDeviceMessageCount": 0,  
    "authenticationType": "sas",  
    "x509Thumbprint": {  
        "primaryThumbprint": null,  
        "secondaryThumbprint": null  
    },  
    "version": 53,  
    "properties": {  
        "desired": { "..."},  
        "reported": {  
            "schemaVersion": "1.0",  
            "version": { "..."},  
            "lastDesiredStatus": { "..."},  
            "runtime": { "..."},  
            "systemModules": {  
                "edgeAgent": { "..."},  
                "edgeHub": { "..."}  
            },  
            "lastDesiredVersion": 5,  
            "modules": {  
                "edgeAgent": { "..."},  
                "edgeHub": { "..."}  
            }  
        }  
    }  
}
```

```
        "SimulatedTemperatureSensor": { "..." }
    },
    "$metadata": { "..." },
    "$version": 48
}
}
```

The JSON can be described in the following sections, starting from the top:

- Metadata - Contains connectivity data. Interestingly, the connection state for the IoT Edge agent is always in a disconnected state: `"connectionState": "Disconnected"`. The reason being the connection state pertains to device-to-cloud (D2C) messages and the IoT Edge agent doesn't send D2C messages.
- Properties - Contains the `desired` and `reported` subsections.
- Properties.desired - (shown collapsed) Expected property values set by the operator in the deployment.json file.
- Properties.reported - Latest property values reported by IoT Edge agent.

Both the `properties.desired` and `properties.reported` sections have a similar structure and contain additional metadata for schema, version, and runtime information. Also included is the `modules` section for any custom modules (such as `SimulatedTemperatureSensor`), and the `systemModules` section for `$edgeAgent` and the `$edgeHub` runtime modules.

By comparing the reported property values against the desired values, you can determine discrepancies and identify disconnections that can help you troubleshoot issues. In doing these comparisons, check the `$lastUpdated` reported value in the `metadata` section for the property you're investigating.

The following properties are important to examine for troubleshooting:

- **exitcode** - Any value other than zero indicates that the module stopped with a failure. However, error codes 137 or 143 are used if a module was intentionally set to a stopped status.
- **lastStartTimeUtc** - Shows the **DateTime** that the container was last started. This value is 0001-01-01T00:00:00Z if the container wasn't started.
- **lastExitTimeUtc** - Shows the **DateTime** that the container last finished. This value is 0001-01-01T00:00:00Z if the container is running and was never stopped.
- **runtimeStatus** - Can be one of the following values:

Value	Description
unknown	Default state until deployment is created.
backoff	The module is scheduled to start but isn't currently running. This value is useful for a module undergoing state changes in restarting. When a failing module is awaiting restart during the cool-off period, the module will be in a backoff state.
running	Indicates that the module is currently running.
unhealthy	Indicates a health-probe check failed or timed out.
stopped	Indicates that the module exited successfully (with a zero exit code).
failed	Indicates that the module exited with a failure exit code (non-zero). The module can transition back to backoff from this state depending on the restart policy in effect. This state can indicate that the module has experienced an unrecoverable error. Failure occurs when the Microsoft Monitoring Agent (MMA) can no longer resuscitate the module, requiring a new deployment.

See [EdgeAgent reported properties](#) for details.

Monitor IoT Edge hub module twin

The following JSON shows the `$edgeHub` module twin in Visual Studio Code with most of the JSON sections collapsed.

JSON

```
{
  "deviceId": "Windows109",
  "moduleId": "$edgeHub",
  "etag": "AAAAAAAAAAU=",
  "deviceEtag": "NzgwNjA1MDU2",
  "status": "enabled",
  "statusUpdateTime": "2001-01-01T00:00:00Z",
  "connectionState": "Connected",
  "lastActivityTime": "2001-01-01T00:00:00Z",
  "cloudToDeviceMessageCount": 0,
  "authenticationType": "sas",
  "x509Thumbprint": {
    "primaryThumbprint": null,
    "secondaryThumbprint": null
  },
  "version": 102,
  "properties": {
    "desired": { "...." },
    "reported": { "...." }
  }
}
```

```
    "reported": {
        "schemaVersion": "1.0",
        "version": { "..."},
        "lastDesiredVersion": 5,
        "lastDesiredStatus": { "..."},
        "clients": {
            "Windows109/SimulatedTemperatureSensor": {
                "status": "Disconnected",
                "lastConnectedTimeUtc": "2020-04-08T21:42:42.1743956Z",
                "lastDisconnectedTimeUtc": "2020-04-09T07:02:42.1398325Z"
            }
        },
        "$metadata": { "..."},
        "$version": 97
    }
}
```

The JSON can be described in the following sections, starting from the top:

- Metadata - Contains connectivity data.
- Properties - Contains the `desired` and `reported` subsections.
- Properties.desired - (shown collapsed) Expected property values set by the operator in the deployment.json file.
- Properties.reported - Latest property values reported by IoT Edge hub.

If you're experiencing issues with your downstream devices, examining this data would be a good place to start.

Monitor custom module twins

The information about the connectivity of your custom modules is maintained in the IoT Edge agent module twin. The module twin for your custom module is used primarily for maintaining data for your solution. The desired properties you defined in your deployment.json file are reflected in the module twin, and your module can update reported property values as needed.

You can use your preferred programming language with the [Azure IoT Hub Device SDKs](#) to update reported property values in the module twin, based on your module's application code. The following procedure uses the Azure SDK for .NET to do this, using code from the [SimulatedTemperatureSensor](#) module:

1. Create an instance of the [ModuleClient](#) with the [CreateFromEnvironmentAysnc](#) method.
2. Get a collection of the module twin's properties with the [GetTwinAsync](#) method.
3. Create a listener (passing a callback) to catch changes to desired properties with the [SetDesiredPropertyUpdateCallbackAsync](#) method.
4. In your callback method, update the reported properties in the module twin with the [UpdateReportedPropertiesAsync](#) method, passing a [TwinCollection](#) of the property values that you want to set.

Access the module twins

You can review the JSON for module twins in the Azure IoT Hub, in Visual Studio Code, and with Azure CLI.

Monitor in Azure IoT Hub

To view the JSON for the module twin:

1. Sign in to the [Azure portal](#)  and navigate to your IoT hub.
2. Select **Devices** under the **Device management** menu.
3. Select the **Device ID** of the IoT Edge device with the modules you want to monitor.
4. Select the module name from the **Modules** tab and then select **Module Identity Twin** from the upper menu bar.

This module identity does not employ an authentication mechanism.

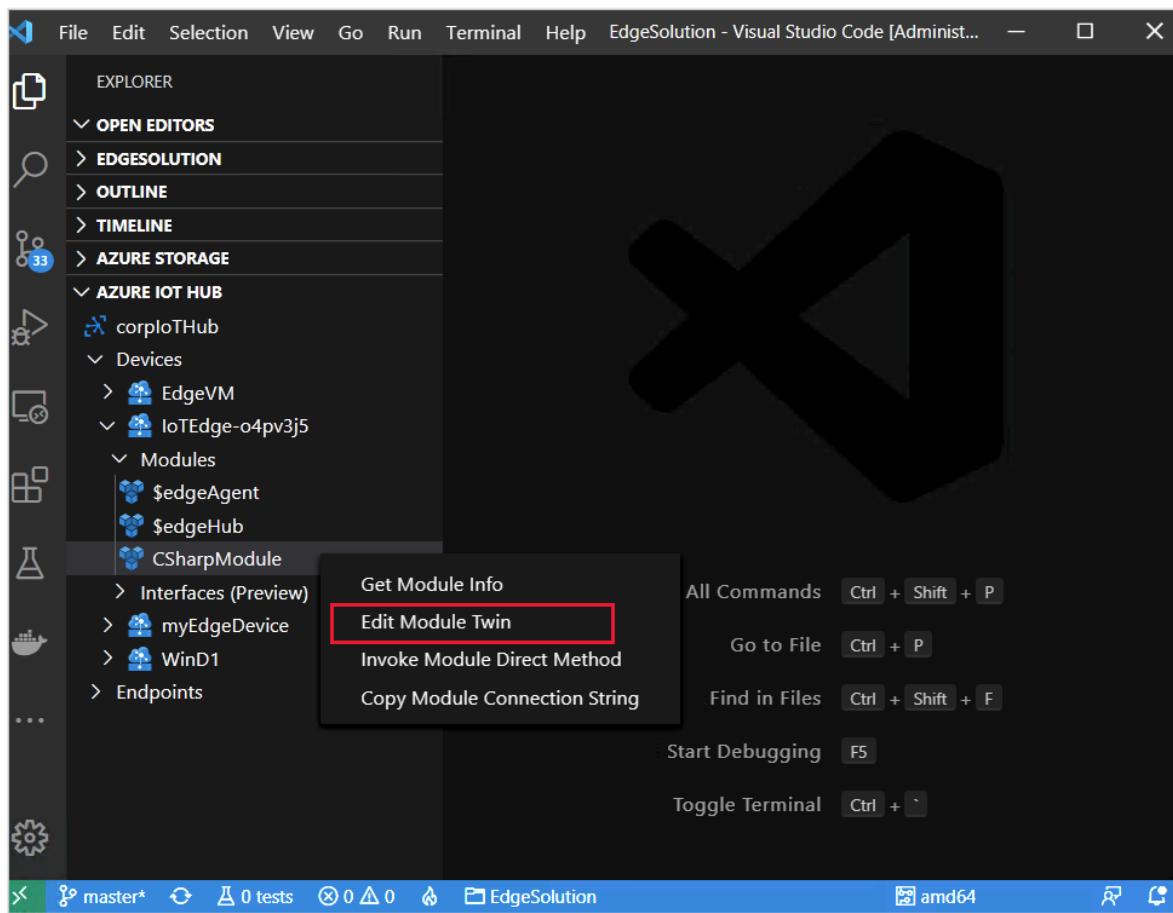
Setting Name	Desired Value	Reported Value
Image URI	mcr.microsoft.com/azureiotedge-hub:1.0	--
Version	--	--
Type	docker	--
Restart Policy	--	--
Last Start Time UTC	N/A	--
Status Description	N/A	--
Runtime Status	N/A	--
Last Exit Time UTC	N/A	--
Exit Code	N/A	--

If you see the message "A module identity doesn't exist for this module", this error indicates that the back-end solution is no longer available that originally created the identity.

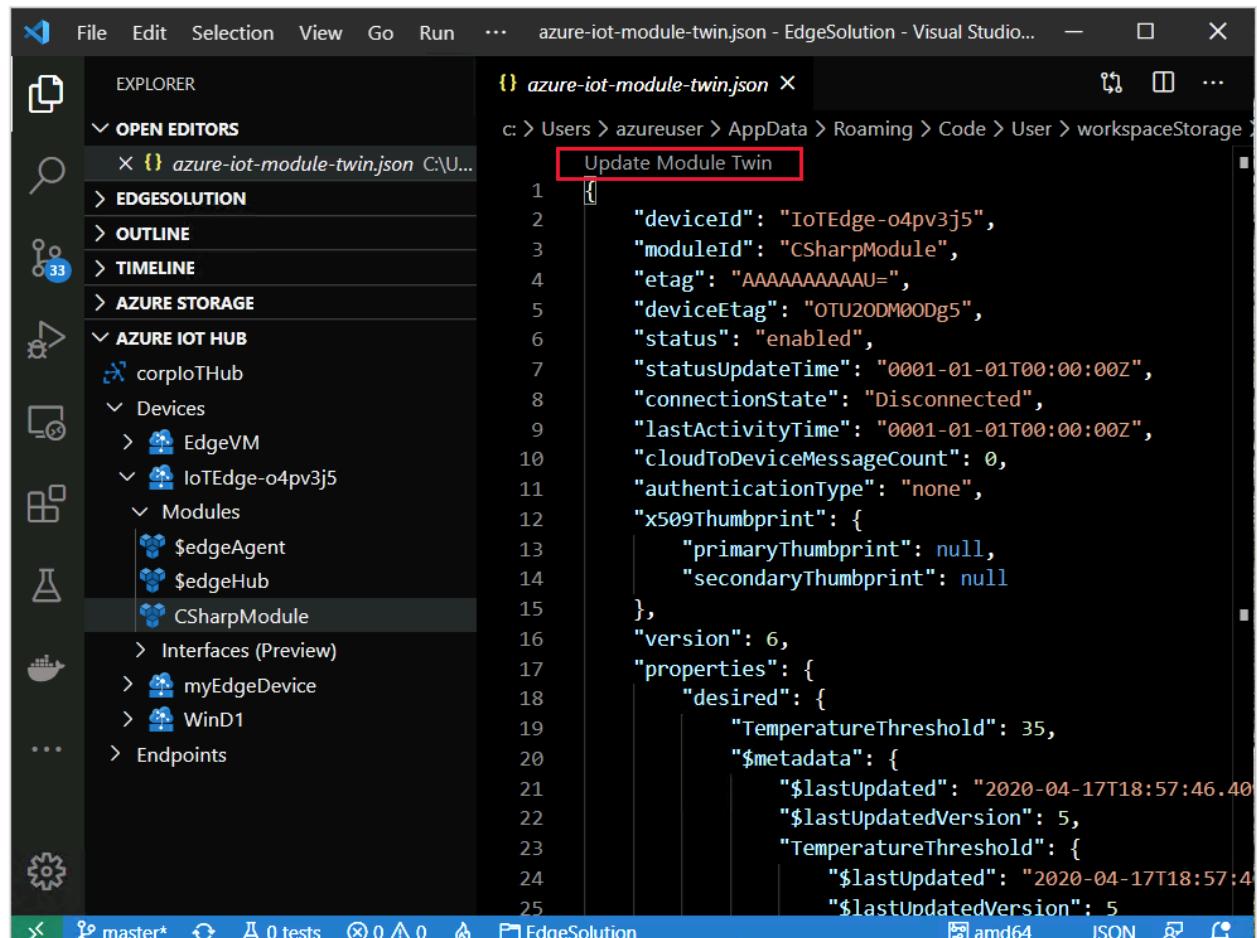
Monitor module twins in Visual Studio Code

To review and edit a module twin:

1. If not already installed, install the [Azure IoT Edge](#) and [Azure IoT Hub](#) extensions. The *Azure IoT Edge tools for Visual Studio Code* extension is in [maintenance mode](#).
2. In the **Explorer**, expand the **Azure IoT Hub**, and then expand the device with the module you want to monitor.
3. Right-click the module and select **Edit Module Twin**. A temporary file of the module twin is downloaded to your computer and displayed in Visual Studio Code.



If you make changes, select **Update Module Twin** above the code in the editor to save changes to your IoT hub.



Monitor module twins in Azure CLI

To see if IoT Edge is running, use the [az iot hub invoke-module-method](#) to ping the IoT Edge agent.

The [az iot hub module-twin](#) structure provides these commands:

- **az iot hub module-twin show** - Show a module twin definition.
- **az iot hub module-twin update** - Update a module twin definition.
- **az iot hub module-twin replace** - Replace a module twin definition with a target JSON.

Tip

To target the runtime modules with CLI commands, you may need to escape the `$` character in the module ID. For example:

Azure CLI

```
az iot hub module-twin show -m '$edgeAgent' -n <hub name> -d <device  
name>
```

Or:

Azure CLI

```
az iot hub module-twin show -m \\$edgeAgent -n <hub name> -d <device  
name>
```

Next steps

Learn how to [communicate with EdgeAgent using built-in direct methods](#).

How to implement IoT Edge observability using monitoring and troubleshooting

Article • 06/03/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

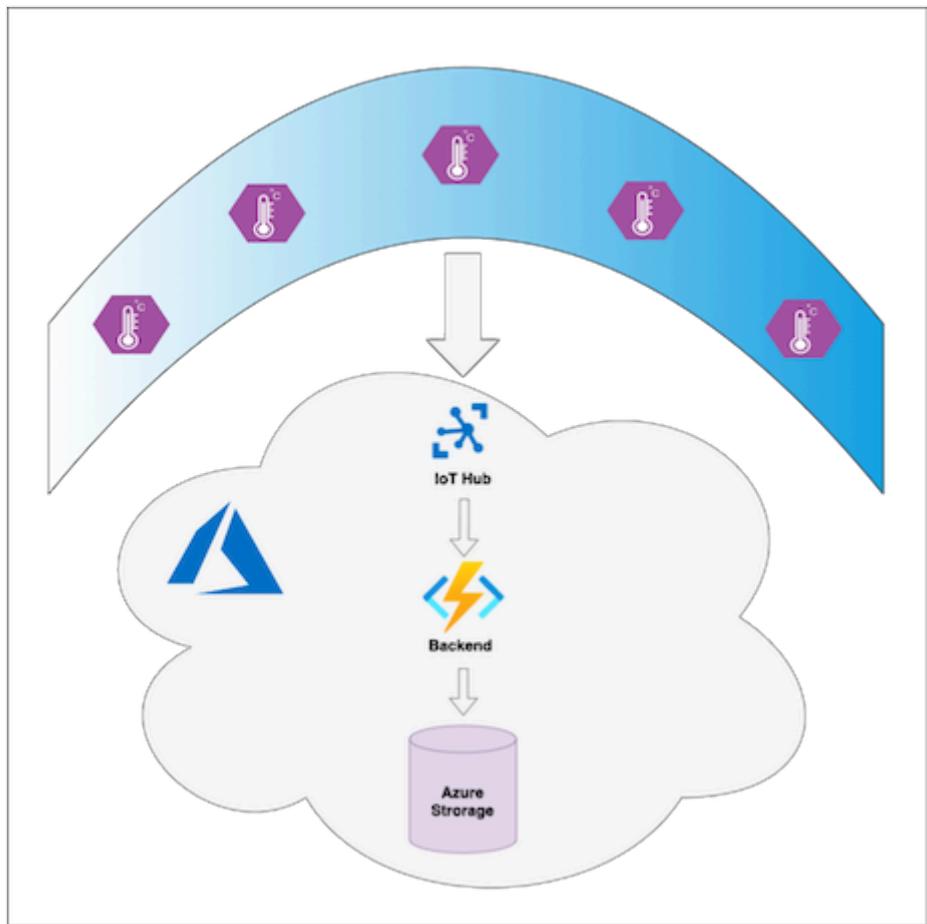
In this article, you'll learn the concepts and techniques of implementing both observability dimensions *measuring and monitoring* and *troubleshooting*. You'll learn about the following topics:

- Define what indicators of the service performance to monitor
- Measure service performance indicators with metrics
- Monitor metrics and detect issues with Azure Monitor workbooks
- Perform basic troubleshooting with the curated workbooks
- Perform deeper troubleshooting with distributed tracing and correlated logs
- Optionally, deploy a sample scenario to Azure to reproduce what you learned

Scenario

In order to go beyond abstract considerations, we'll use a *real-life* scenario collecting ocean surface temperatures from sensors into Azure IoT.

La Niña



The La Niña service measures surface temperature in Pacific Ocean to predict La Niña winters. There are many buoys in the ocean with IoT Edge devices that send the surface temperature to Azure Cloud. The telemetry data with the temperature is pre-processed by a custom module on the IoT Edge device before sending it to the cloud. In the cloud, the data is processed by backend Azure Functions and saved to Azure Blob Storage. The clients of the service (ML inference workflows, decision making systems, various UIs, etc.) can pick up messages with temperature data from the Azure Blob Storage.

Measuring and monitoring

Let's build a measuring and monitoring solution for the La Niña service focusing on its business value.

What do we measure and monitor

To understand what we're going to monitor, we must understand what the service actually does and what the service clients expect from the system. In this scenario, the expectations of a common La Niña service consumer may be categorized by the following factors:

- **Coverage.** The data is coming from most installed buoys
- **Freshness.** The data coming from the buoys is fresh and relevant

- **Throughput.** The temperature data is delivered from the buoys without significant delays
- **Correctness.** The ratio of lost messages (errors) is small

The satisfaction regarding these factors means that the service works according to the client's expectations.

The next step is to define instruments to measure values of these factors. This job can be done by the following Service Level Indicators (SLI):

[\[+\] Expand table](#)

Service Level Indicator	Factors
Ratio of on-line devices to the total number of devices	Coverage
Ratio of devices reporting frequently to the number of reporting devices	Freshness, Throughput
Ratio of devices successfully delivering messages to the total number of devices	Correctness
Ratio of devices delivering messages fast to the total number of devices	Throughput

With that done, we can apply a sliding scale on each indicator and define exact threshold values that represent what it means for the client to be "satisfied". For this scenario, we select sample threshold values as laid out in the table below with formal Service Level Objectives (SLOs):

[\[+\] Expand table](#)

Service Level Objective	Factor
90% of devices reported metrics no longer than 10 mins ago (were online) for the observation interval	Coverage
95% of online devices send temperature 10 times per minute for the observation interval	Freshness, Throughput
99% of online devices deliver messages successfully with less than 5% of errors for the observation interval	Correctness
95% of online devices deliver 90th percentile of messages within 50 ms for the observation interval	Throughput

SLOs definition must also describe the approach of how the indicator values are measured:

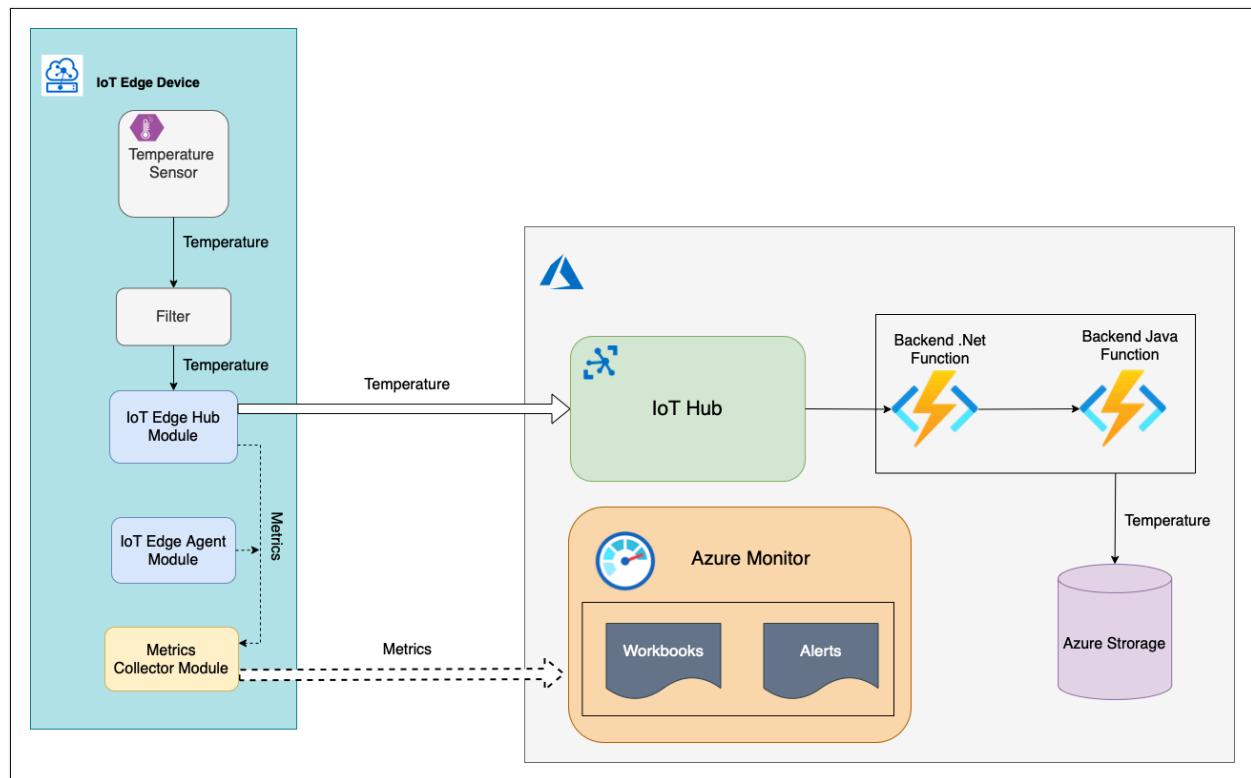
- Observation interval: 24 hours. SLO statements have been true for the last 24 hours. Which means that if an SLI goes down and breaks a corresponding SLO, it will take 24 hours after the SLI has been fixed to consider the SLO good again.
- Measurements frequency: 5 minutes. We do the measurements to evaluate SLI values every 5 minutes.
- What is measured: interaction between IoT Device and the cloud, further consumption of the temperature data is out of scope.

How do we measure

At this point, it's clear what we're going to measure and what threshold values we're going to use to determine if the service performs according to the expectations.

It's a common practice to measure service level indicators, like the ones we've defined, by the means of **metrics**. This type of observability data is considered to be relatively small in values. It's produced by various system components and collected in a central observability backend to be monitored with dashboards, workbooks and alerts.

Let's clarify what components the La Niña service consists of:



There is an IoT Edge device with **Temperature Sensor** custom module (C#) that generates some temperature value and sends it upstream with a telemetry message. This message is routed to another custom module **Filter** (C#). This module checks the received temperature against a threshold window (0-100 degrees Celsius). If the

temperature is within the window, the FilterModule sends the telemetry message to the cloud.

In the cloud, the message is processed by the backend. The backend consists of a chain of two Azure Functions and storage account. Azure .NET Function picks up the telemetry message from the IoT Hub events endpoint, processes it and sends it to Azure Java Function. The Java function saves the message to the storage account blob container.

An IoT Hub device comes with system modules `edgeHub` and `edgeAgent`. These modules expose through a Prometheus endpoint [a list of built-in metrics](#). These metrics are collected and pushed to Azure Monitor Log Analytics service by the [metrics collector module](#) running on the IoT Edge device. In addition to the system modules, the `Temperature Sensor` and `Filter` modules can be instrumented with some business specific metrics too. However, the service level indicators that we've defined can be measured with the built-in metrics only. So, we don't really need to implement anything else at this point.

In this scenario, we have a fleet of 10 buoys. One of the buoys is intentionally set up to malfunction so that we can demonstrate the issue detection and the follow-up troubleshooting.

How do we monitor

We're going to monitor Service Level Objectives (SLO) and corresponding Service Level Indicators (SLI) with Azure Monitor Workbooks. This scenario deployment includes the *La Nina SLO/SLI* workbook assigned to the IoT Hub.

Dashboard > iothub-a766ea18

iothub-a766ea18 | Workbooks | Gallery

IoT Hub

Search (Cmd +/)

New Refresh Feedback Help

Identity Shared access policies Networking Certificates

Private and favorite Workbooks are deprecated and will be removed in a future release.

All Workbooks Public Templates My Templates

Filter by name or category

Subscriptions

Defender for IoT

- Overview
- Security Alerts
- Recommendations
- Settings

Monitoring

- Alerts
- Metrics
- Diagnostic settings
- Logs

Workbooks

Automation

- Tasks (preview)
- Export template

Quick start

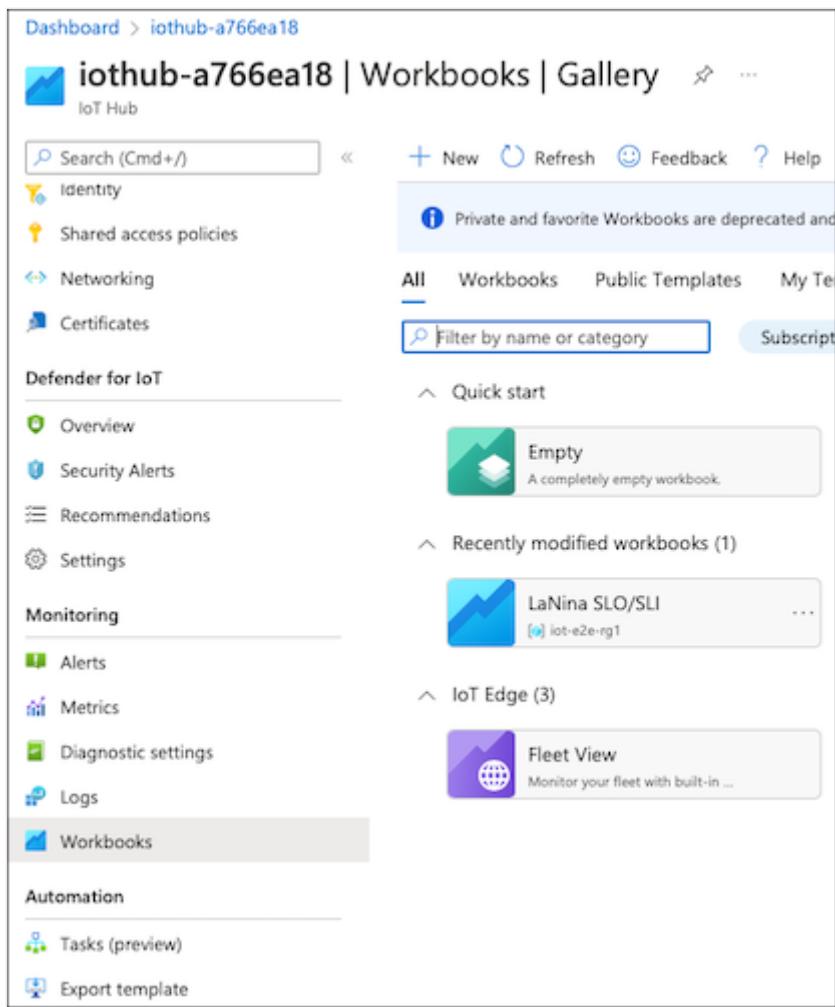
Empty A completely empty workbook.

Recently modified workbooks (1)

LaNina SLO/SLI [iot-e2e-rg1] ...

IoT Edge (3)

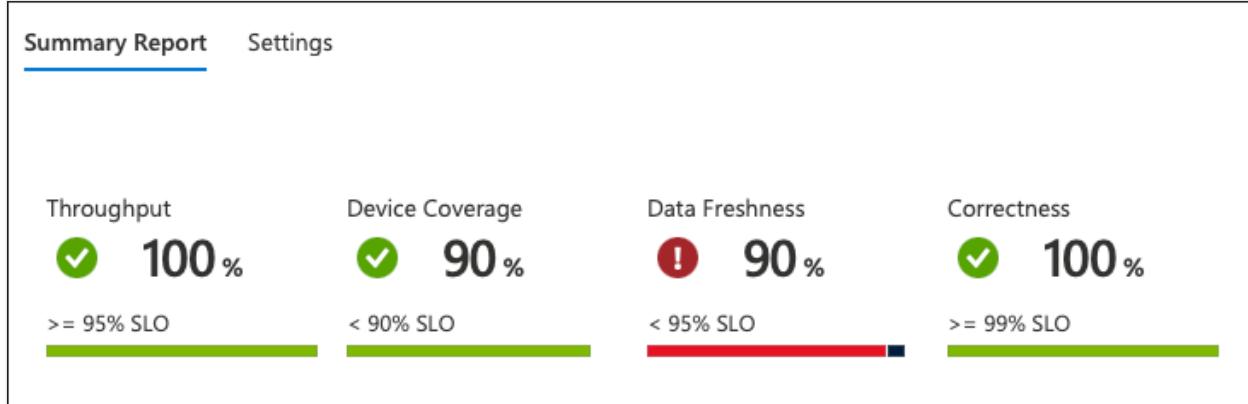
Fleet View Monitor your fleet with built-in ...



To achieve the best user experience the workbooks are designed to follow the *glance* -> *scan* -> *commit* concept:

Glance

At this level, we can see the whole picture at a single glance. The data is aggregated and represented at the fleet level:



From what we can see, the service is not functioning according to the expectations. There is a violation of the *Data Freshness* SLO. Only 90% of the devices send the data frequently, and the service clients expect 95%.

All SLO and threshold values are configurable on the workbook settings tab:

The screenshot shows the 'Settings' tab of a workbook. It includes sections for 'Observation interval' (Last 24 hours), 'Device Offline Threshold (minutes)' (10), 'Coverge SLO value (%)' (90), 'Latency Threshold (milliseconds)' (50), 'Throughput SLO value (%)' (95), 'Message Frequency Threshold (messaages per minute)' (10), 'Freshness SLO value (%)' (95), 'Errors Threshold (%)' (5), and 'Correctness SLO value (%)' (99).

Scan

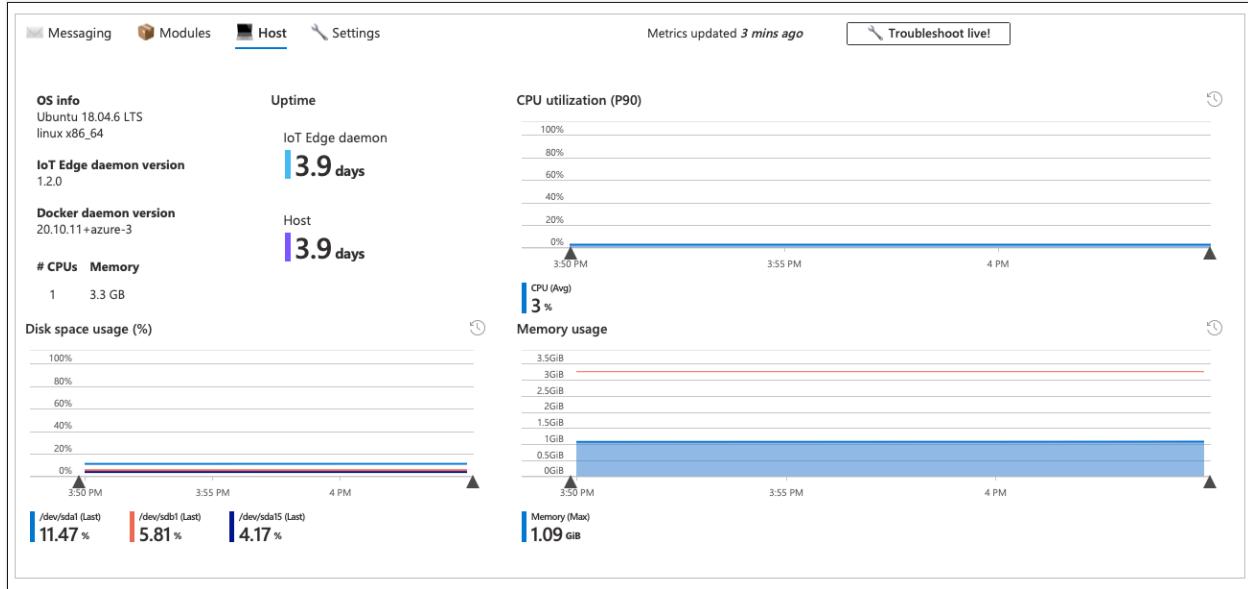
By clicking on the violated SLO, we can drill down to the *scan* level and see how the devices contribute to the aggregated SLI value.

GroupName	Device	minMessageFrequency	time_range	maxMessageFrequency	p90MessageFrequency	Trend
① Rare (1)	iotedgevm1-56ed9473	4	3/14 22:56 - 3/14 23:1	5.8	5.8	
✓ Frequent (9)	iotedgevm9-56ed9473	11.8	3/14 22:54 - 3/14 22:59	12	12	
	iotedgevm10-56ed9473	11.8	3/14 22:55 - 3/14 23:0	12	12	
	iotedgevm8-56ed9473	12	3/14 22:54 - 3/14 22:59	12	12	
	iotedgevm6-56ed9473	12	3/14 22:54 - 3/14 22:59	12	12	
	iotedgevm7-56ed9473	12	3/14 22:55 - 3/14 23:0	12	12	
	iotedgevm2-56ed9473	12	3/14 22:54 - 3/14 22:59	12	12	
	iotedgevm5-56ed9473	12	3/14 22:53 - 3/14 22:58	12	12	
✓ Frequent	interneumaLcharQAT2	12	3/14 22:56 - 3/14 23:1	12	12	

There is a single device (out of 10) that sends the telemetry data to the cloud "rarely". In our SLO definition, we've stated that "frequently" means at least 10 times per minute. The frequency of this device is way below that threshold.

Commit

By clicking on the problematic device, we're drilling down to the *commit* level. This is a curated workbook *Device Details* that comes out of the box with IoT Hub monitoring offering. The *La Nina SLO/SLI* workbook reuses it to bring the details of the specific device performance.



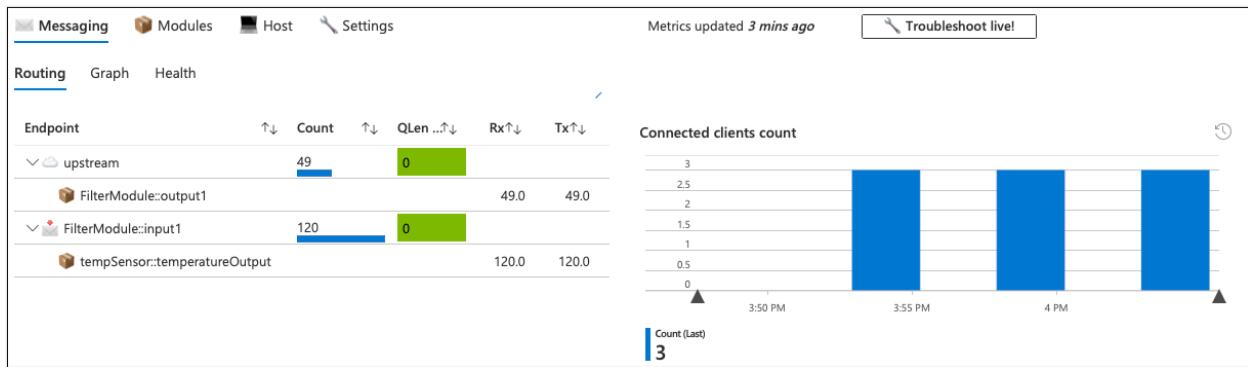
Troubleshooting

Measuring and monitoring lets us observe and predict the system behavior, compare it to the defined expectations and ultimately detect existing or potential issues. The *troubleshooting*, on the other hand, lets identify and locate the cause of the issue.

Basic troubleshooting

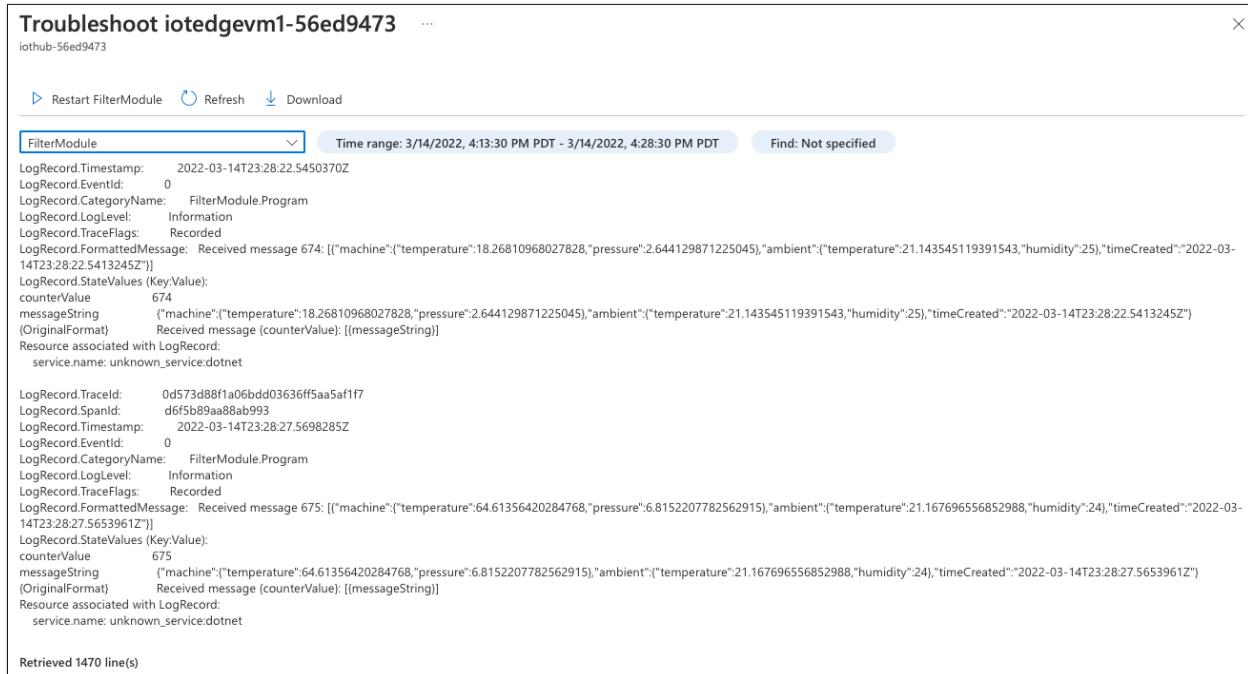
The *commit* level workbook gives a lot of detailed information about the device health. That includes resources consumption at the module and device level, message latency, frequency, QLen, etc. In many cases, this information may help locate the root of the issue.

In this scenario, all parameters of the trouble device look normal and it's not clear why the device sends messages less frequent than expected. This fact is also confirmed by the *messaging* tab of the device-level workbook:



The `Temperature Sensor` (`tempSensor`) module produced 120 telemetry messages, but only 49 of them went upstream to the cloud.

The first step we want to do is to check the logs produced by the `Filter` module. Select `Troubleshoot live!` then select the `Filter` module.

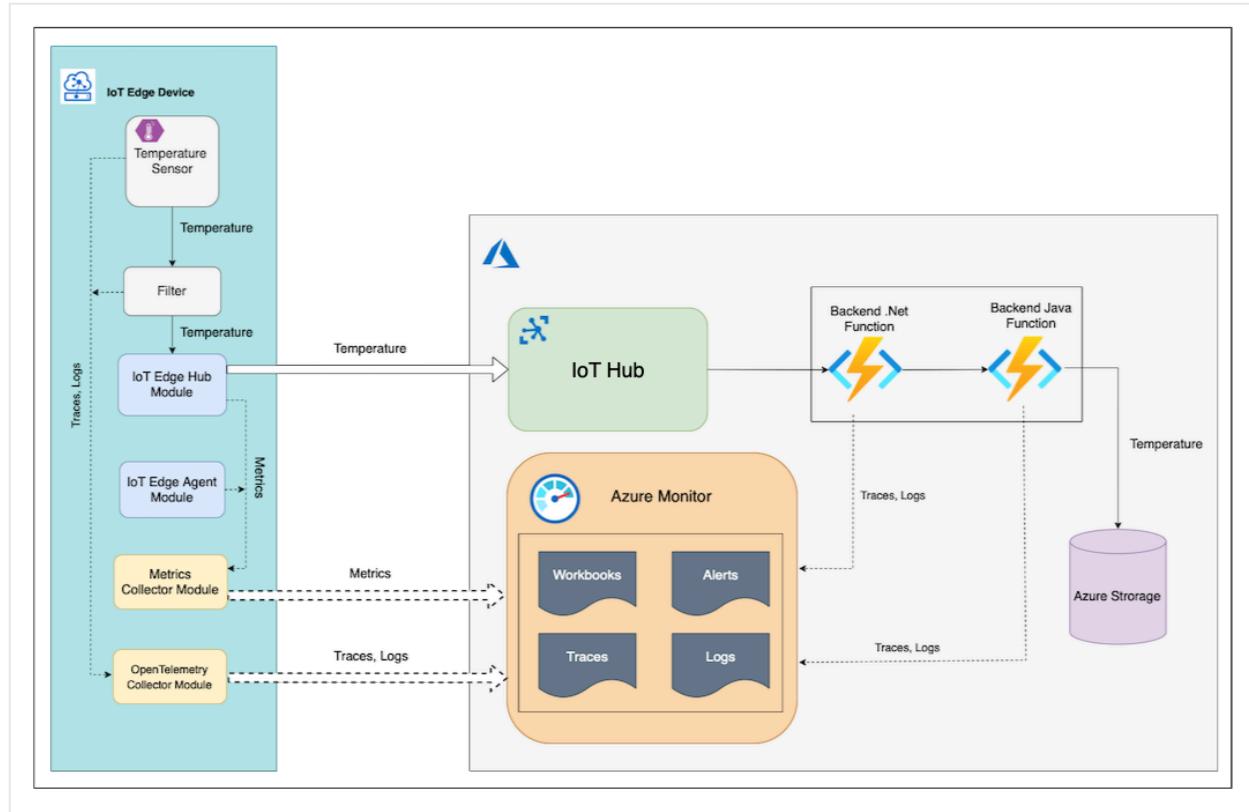


Analysis of the module logs doesn't discover the issue. The module receives messages, there are no errors. Everything looks good here.

Deep troubleshooting

There are two observability instruments serving the deep troubleshooting purposes: *traces* and *logs*. In this scenario, traces show how a telemetry message with the ocean surface temperature is traveling from the sensor to the storage in the cloud, what is invoking what and with what parameters. Logs give information on what is happening inside each system component during this process. The real power of *traces* and *logs* comes when they're correlated. With that it's possible to read the logs of a specific system component, such as a module on IoT device or a backend function, while it was processing a specific telemetry message.

The La Niña service uses [OpenTelemetry](#) to produce and collect traces and logs in Azure Monitor.



IoT Edge modules `Temperature Sensor` and `Filter` export the logs and tracing data via OTLP (OpenTelemetry Protocol) to the [OpenTelemetryCollector](#) module, running on the same edge device. The `OpenTelemetryCollector` module, in its turn, exports logs and traces to Azure Monitor Application Insights service.

The Azure .NET Function sends the tracing data to Application Insights with [Azure Monitor Open Telemetry direct exporter](#). It also sends correlated logs directly to Application Insights with a configured ILogger instance.

The Java backend function uses [OpenTelemetry auto-instrumentation Java agent](#) to produce and export tracing data and correlated logs to the Application Insights instance.

By default, IoT Edge modules on the devices of the La Niña service are configured to not produce any tracing data and the `logging level` is set to `Information`. The amount of produced tracing data is regulated by a [ratio based sampler](#). The sampler is configured with a desired [probability](#) of a given activity to be included in a trace. By default, the probability is set to 0. With that in place, the devices don't flood the Azure Monitor with the detailed observability data if it's not requested.

We've analyzed the `Information` level logs of the `Filter` module and realized that we need to dive deeper to locate the cause of the issue. We're going to update properties

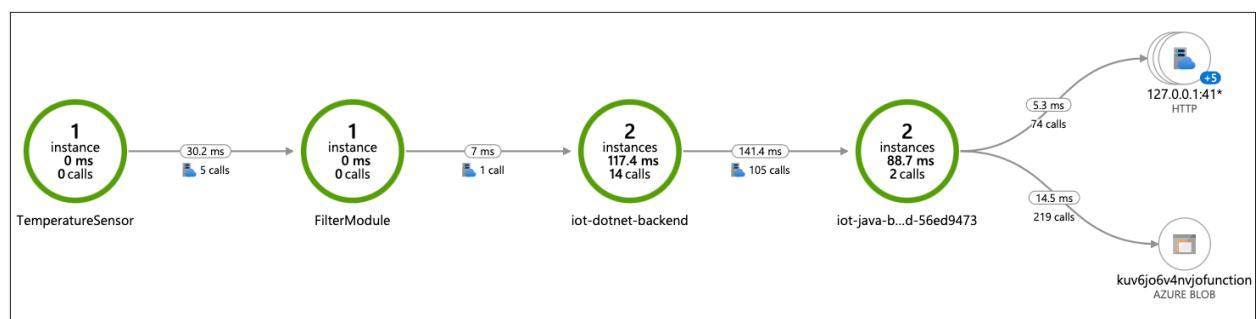
in the `Temperature Sensor` and `Filter` module twins and increase the `loggingLevel` to `Debug` and change the `traceSampleRatio` from `0` to `1`:

```
"properties": {
    "desired": {
        ...
        "loggingLevel": "Debug",
        "traceSampleRatio": "1"
    },
    "reported": {
        "loggingLevel": "Information",
        "traceSampleRatio": "0",
        "$metadata": {
            ...
        }
    }
}
```

With that in place, we have to restart the `Temperature Sensor` and `Filter` modules:

The screenshot shows the Azure IoT Hub Troubleshoot interface for the device `iohub-56ed9473`. The top navigation bar includes `Restart FilterModule`, `Refresh`, and `Download` buttons. Below the navigation is a search bar with `FilterModule` selected and a time range filter set to `Time range: Since 15 minutes`.

In a few minutes, the traces and detailed logs will arrive to Azure Monitor from the trouble device. The entire end-to-end message flow from the sensor on the device to the storage in the cloud will be available for monitoring with *application map* in Application Insights:



From this map we can drill down to the traces and we can see that some of them look normal and contain all the steps of the flow, and some of them, are short, so nothing happens after the `Filter` module.

End-to-end transaction	
Operation ID: 7262faab7487231fcf06a9223cf5e873	
EVENT	RES.
OTHER SendTemeperature	0
FilterModule FilterTemperature	0
OTHER Upstream	0
iot-dotnet-backend ProcessedInFunc	0
OTHER Invoke Java	
iot-java-backend-56ed9473 prc 200	
AzureBlobStorageCont.getPro	
kuv6jo6v4nvjofunction A: 200	
127.0.0.1:41392 GET /MSI/tol 200	
AzureBlobStorageBloc.upload	
kuv6jo6v4nvjofunction A: 201	
AzureBlobStorageBlob.setMet	
kuv6jo6v4nvjofunction A: 200	
Telemetry type	request

End-to-end transaction	
Operation ID: 31e8cc25367ac9176fef088fb7a8c73e	
EVENT	RES.
OTHER SendTemeperature	0
FilterModule FilterTemperature	0
Request Properties	
Event time	3/15/2022, 12:41:50.241 PM (Local time)
Name	FilterTemperature
Response code	0
Successful request	true
Response time	5.4 ms
Operation Id	7262faab7487231fcf06a9223cf5e873
Parent Id	d8c4dada69ceb507
Id	e1a8ec2c30d40197
Telemetry type	request

Let's analyze one of those short traces and find out what was happening in the `Filter` module, and why it didn't send the message upstream to the cloud.

Our logs are correlated with the traces, so we can query logs specifying the `TraceId` and `SpanId` to retrieve logs corresponding exactly to this execution instance of the `Filter` module:

```

1 traces
2 | where
3   customDimensions['TraceId'] == '31e8cc25367ac9176fef088fb7a8c73e'
4   and
5   customDimensions['SpanId'] == '1539876b1d97fc27'

```

Results	
timestamp [UTC]	message
> 3/15/2022, 7:42:40.507 PM	Received message 763: [{"machine": {"temperature": 70.46505276601066, "pressure": 100}], status: 'Message didn't pass threshold 30-70'
3/15/2022, 7:42:40.508 ...	Message didn't pass threshold 30-70
timestamp [UTC]	2022-03-15T19:42:40.5085788Z
message	Message didn't pass threshold 30-70
severityLevel	0
itemType	trace
customDimensions	{"CategoryName": "FilterModule.Program", "TraceId": "31e8cc25367ac9176fef088fb7a8c73e", "SpanId": "1539876b1d97fc27", "System": {"Log": "Azure-LogAnalytics", "Source": "FilterModule", "Type": "Trace"}, "Time": "2022-03-15T19:42:40.5085788Z", "Version": "1.0"} operation_Id
operation_Id	31e8cc25367ac9176fef088fb7a8c73e

The logs show that the module received a message with 70.465-degrees temperature. But the filtering threshold configured on this device is 30 to 70. So the message simply didn't pass the threshold. Apparently, this specific device was configured wrong. This is the cause of the issue we detected while monitoring the La Niña service performance with the workbook.

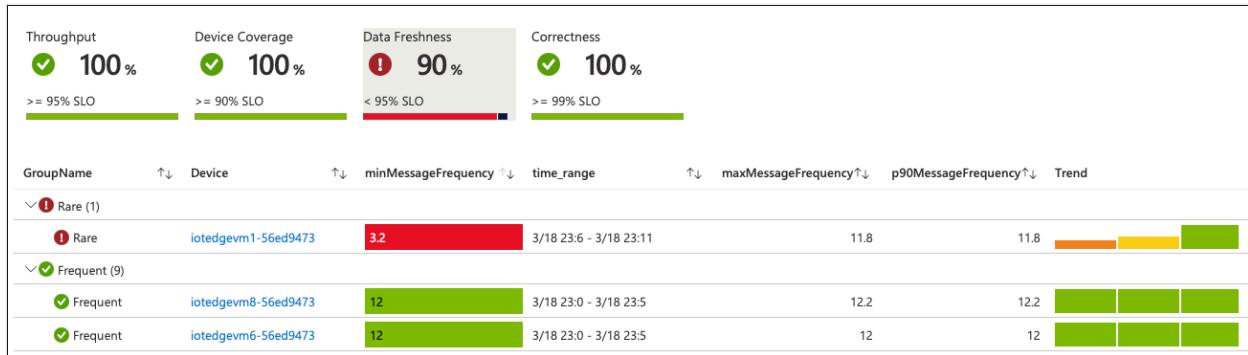
Let's fix the `Filter` module configuration on this device by updating properties in the module twin. We also want to reduce back the `loggingLevel` to `Information` and `traceSampleRatio` to `0`:

```

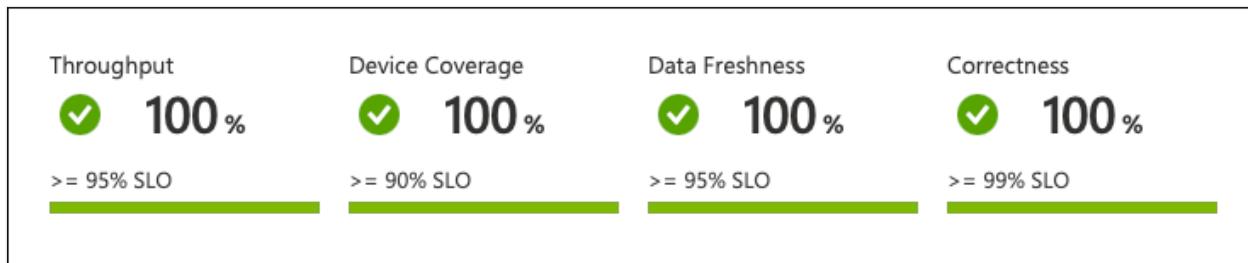
"properties": {
    "desired": {
        "minTemperatureThreshold": 0,
        "maxTemperatureThreshold": 100,
        "loggingLevel": "Information",
        "traceSampleRatio": "0",
        "$metadata": {

```

Having done that, we need to restart the module. In a few minutes, the device reports new metric values to Azure Monitor. It reflects in the workbook charts:



We see that the message frequency on the problematic device got back to normal. The overall SLO value will become green again, if nothing else happens, in the configured observation interval:



Try the sample

At this point, you might want to deploy the scenario sample to Azure to reproduce the steps and play with your own use cases.

In order to successfully deploy this solution, you need the following:

- [PowerShell](#).
- [Azure CLI](#).
- An Azure account with an active subscription. [Create one for free ↗](#).

1. Clone the [IoT Elms ↗](#) repository.

sh

```
git clone https://github.com/Azure-Samples/iotedge-logging-and-monitoring-solution.git
```

2. Open a PowerShell console and run the `deploy-e2e-tutorial.ps1` script.

```
PowerShell
```

```
./Scripts/deploy-e2e-tutorial.ps1
```

Next steps

In this article, you have set up a solution with end-to-end observability capabilities for monitoring and troubleshooting. The common challenge in such solutions for IoT systems is delivering observability data from the devices to the cloud. The devices in this scenario are supposed to be online and have a stable connection to Azure Monitor, which is not always the case in real life.

Advance to follow up articles such as [Distributed Tracing with IoT Edge](#) with the recommendations and techniques to handle scenarios when the devices are normally offline or have limited or restricted connection to the observability backend in the cloud.

Communicate with edgeAgent using built-in direct methods

Article • 02/23/2023

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Monitor and manage IoT Edge deployments by using the direct methods included in the IoT Edge agent module. Direct methods are implemented on the device, and then can be invoked from the cloud. The IoT Edge agent includes direct methods that help you monitor and manage your IoT Edge devices remotely.

For more information about direct methods, how to use them, and how to implement them in your own modules, see [Understand and invoke direct methods from IoT Hub](#).

The names of these direct methods are handled case-insensitive.

Ping

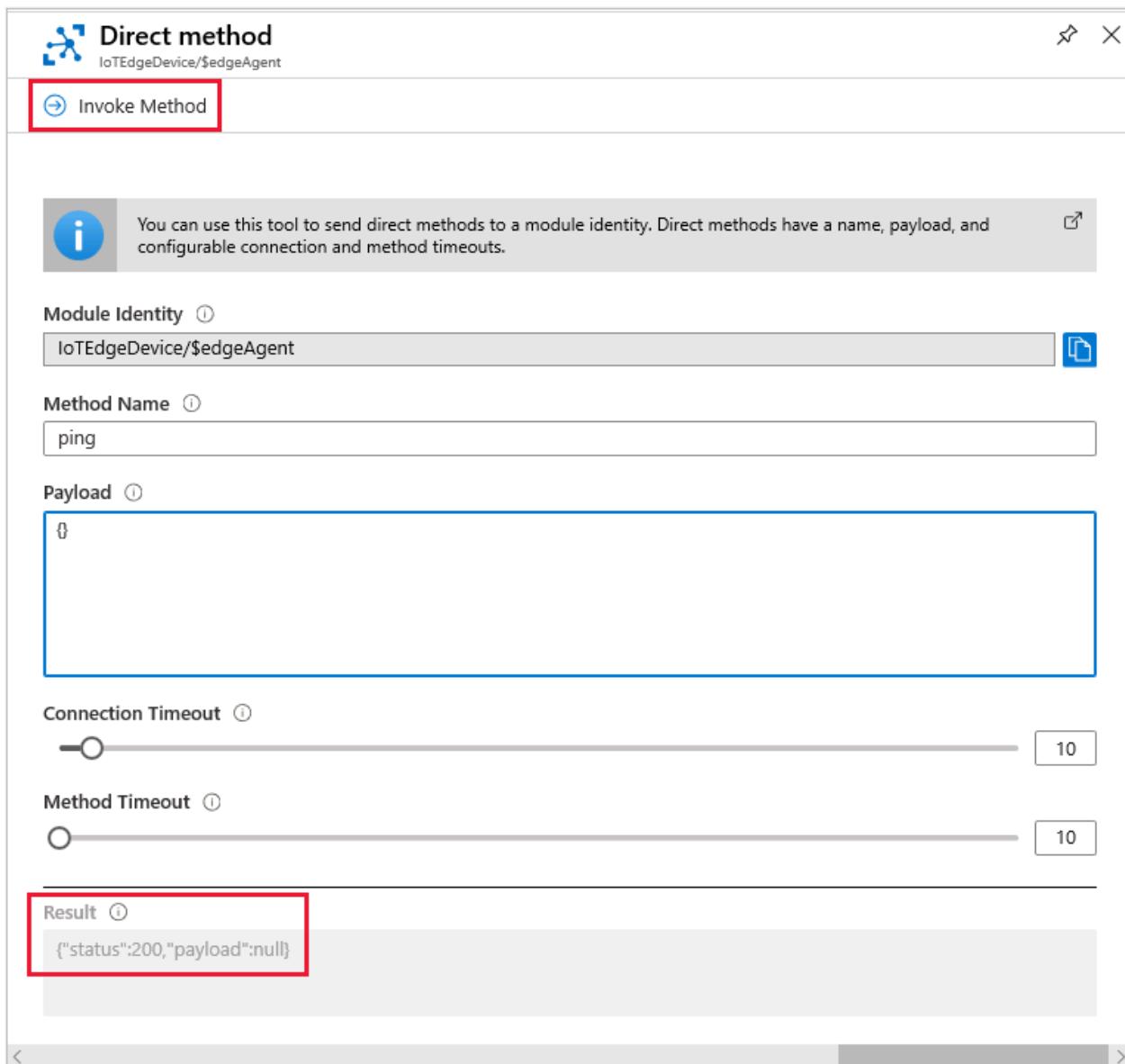
The **ping** method is useful for checking whether IoT Edge is running on a device, or whether the device has an open connection to IoT Hub. Use this direct method to ping the IoT Edge agent and get its status. A successful ping returns an empty payload and "status": 200.

For example:

Azure CLI

```
az iot hub invoke-module-method --method-name 'ping' -n <hub name> -d <device name> -m '$edgeAgent'
```

In the Azure portal, invoke the method with the method name `ping` and an empty JSON payload `{}`.



Restart module

The **RestartModule** method allows for remote management of modules running on an IoT Edge device. If a module is reporting a failed state or other unhealthy behavior, you can trigger the IoT Edge agent to restart it. A successful restart command returns an empty payload and "status": 200.

The `RestartModule` method is available in IoT Edge version 1.0.9 and later.

Tip

The IoT Edge troubleshooting page in the Azure portal provides a simplified experience for restarting modules. For more information, see [Monitor and troubleshoot IoT Edge devices from the Azure portal](#).

You can use the `RestartModule` direct method on any module running on an IoT Edge device, including the `edgeAgent` module itself. However, if you use this direct method to shut down the `edgeAgent`, you won't receive a success result since the connection is disrupted while the module restarts.

For example:

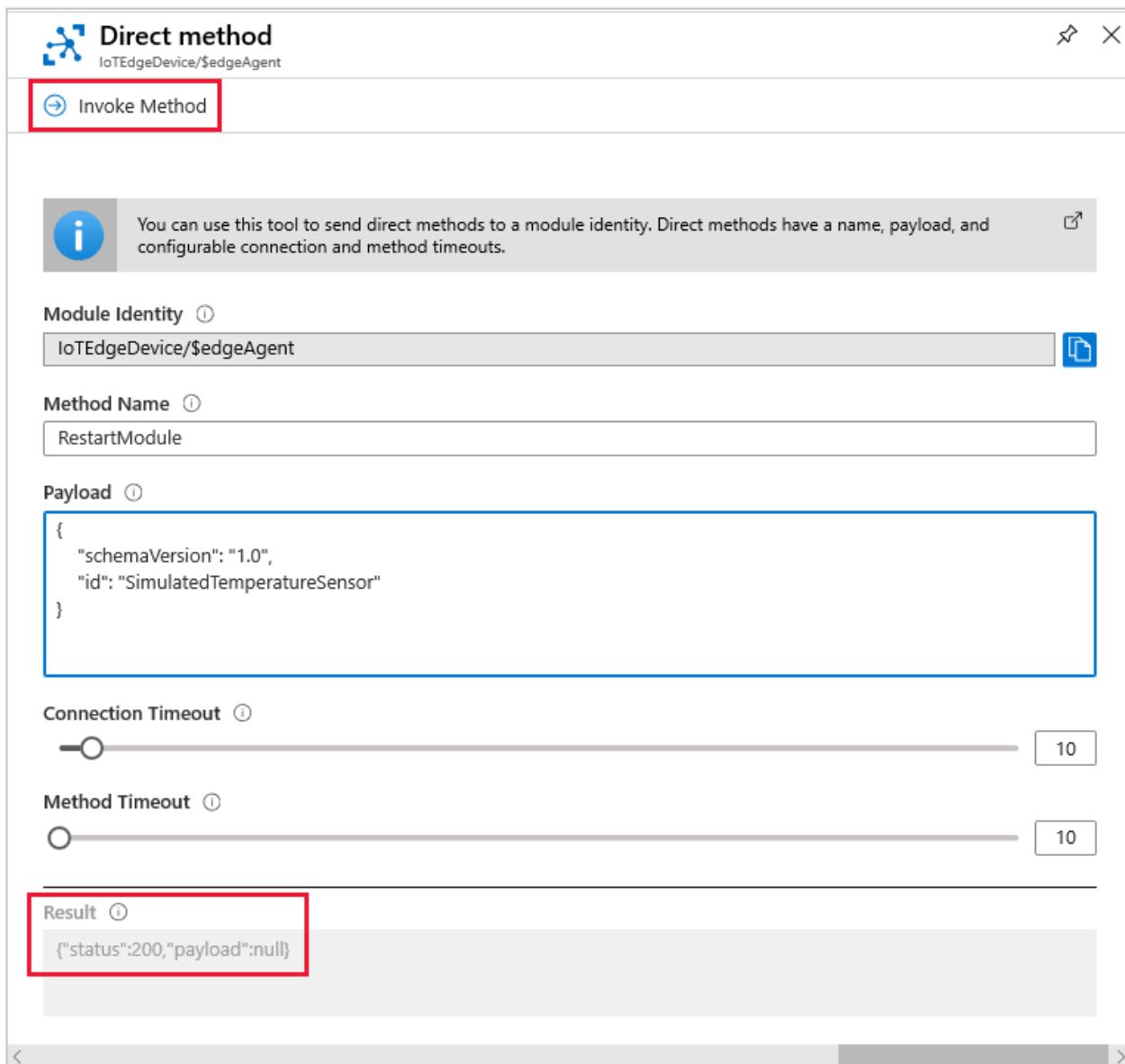
Azure CLI

```
az iot hub invoke-module-method --method-name 'RestartModule' -n <hub name>
-d <device name> -m '$edgeAgent' --method-payload \
'
{
    "schemaVersion": "1.0",
    "id": "<module name>"
}
'
```

In the Azure portal, invoke the method with the method name `RestartModule` and the following JSON payload:

JSON

```
{
    "schemaVersion": "1.0",
    "id": "<module name>"
}
```



Diagnostic direct methods

- [GetModuleLogs](#): Retrieve module logs inline in the response of the direct method.
- [UploadModuleLogs](#): Retrieve module logs and upload them to Azure Blob Storage.
- [UploadSupportBundle](#): Retrieve module logs using a support bundle and upload a zip file to Azure Blob Storage.
- [GetTaskStatus](#): Check on the status of an upload logs or support bundle request.

These diagnostic direct methods are available as of the 1.0.10 release.

Next steps

[Properties of the IoT Edge agent and IoT Edge hub module twins](#)

Retrieve logs from IoT Edge deployments

Article • 06/04/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Retrieve logs from your IoT Edge deployments without needing physical or SSH access to the device by using the direct methods included in the IoT Edge agent module. Direct methods are implemented on the device, and then can be invoked from the cloud. The IoT Edge agent includes direct methods that help you monitor and manage your IoT Edge devices remotely. The direct methods discussed in this article are generally available with the 1.0.10 release.

For more information about direct methods, how to use them, and how to implement them in your own modules, see [Understand and invoke direct methods from IoT Hub](#).

The names of these direct methods are handled case-sensitive.

Recommended logging format

While not required, for best compatibility with this feature, the recommended logging format is:

```
<{Log Level}> {Timestamp} {Message Text}
```

{Timestamp} should be formatted as yyyy-MM-dd HH:mm:ss.fff zzz, and {Log Level} should use the following table, which derives its severity levels from the [Severity code in the Syslog standard](#).

 Expand table

Value	Severity
0	Emergency
1	Alert
2	Critical
3	Error
4	Warning
5	Notice
6	Informational
7	Debug

The [Logger class in IoT Edge](#) serves as a canonical implementation.

Retrieve module logs

Use the **GetModuleLogs** direct method to retrieve the logs of an IoT Edge module.

Tip

Use the `since` and `until` filter options to limit the range of logs retrieved. Calling this direct method without bounds retrieves all the logs which may be large, time consuming, or costly.

The IoT Edge troubleshooting page in the Azure portal provides a simplified experience for viewing module logs. For more information, see [Monitor and troubleshoot IoT Edge devices from the Azure portal](#).

This method accepts a JSON payload with the following schema:

JSON

```
{
  "schemaVersion": "1.0",
  "items": [
    {
      "id": "regex string",
      "filter": {
        "tail": "int",
        "since": "string",
        "until": "string",
      }
    }
  ]
}
```

```

        "loglevel": "int",
        "regex": "regex string"
    }
],
"encoding": "gzip/none",
"contentType": "json/text"
}

```

[\[\] Expand table](#)

Name	Type	Description
schemaVersion	string	Set to <code>1.0</code>
items	JSON array	An array with <code>id</code> and <code>filter</code> tuples.
id	string	A regular expression that supplies the module name. It can match multiple modules on an edge device. .NET Regular Expressions format is expected. In case there are multiple items whose ID matches the same module, only the filter options of the first matching ID is applied to that module.
filter	JSON section	Log filters to apply to the modules matching the <code>id</code> regular expression in the tuple.
tail	integer	Number of log lines in the past to retrieve starting from the latest. OPTIONAL.
since	string	Only return logs since this time, as an rfc3339 timestamp, UNIX timestamp, or a duration (days (d) hours (h) minutes (m)). For example, a duration for one day, 12 hours, and 30 minutes can be specified as <i>1 day 12 hours 30 minutes</i> or <i>1d 12h 30m</i> . If both <code>tail</code> and <code>since</code> are specified, the logs are retrieved using the <code>since</code> value first. Then, the <code>tail</code> value is applied to the result, and the final result is returned. OPTIONAL.
until	string	Only return logs before the specified time, as an rfc3339 timestamp, UNIX timestamp, or duration (days (d) hours (h) minutes (m)). For example, a duration 90 minutes can be specified as <i>90 minutes</i> or <i>90m</i> . If both <code>tail</code> and <code>since</code> are specified, the logs are retrieved using the <code>since</code> value first. Then, the <code>tail</code> value is applied to the result, and the final result is returned. OPTIONAL.
loglevel	integer	Filter log lines equal to specified log level. Log lines should follow recommended logging format and use Syslog severity level standard. Should you need to filter by multiple log level severity values, then rely on regex matching, provided the module follows

Name	Type	Description
		some consistent format when logging different severity levels. OPTIONAL.
regex	string	Filter log lines that have content that match the specified regular expression using .NET Regular Expressions format. OPTIONAL.
encoding	string	Either <code>gzip</code> or <code>none</code> . Default is <code>none</code> .
contentType	string	Either <code>json</code> or <code>text</code> . Default is <code>text</code> .

ⓘ Note

If the logs content exceeds the response size limit of direct methods, which is currently 128 KB, the response returns an error.

A successful retrieval of logs returns a "status": 200 followed by a payload containing the logs retrieved from the module, filtered by the settings you specify in your request.

For example:

Azure CLI

```
az iot hub invoke-module-method --method-name 'GetModuleLogs' -n <hub name>
-d <device id> -m '$edgeAgent' --method-payload \
'
{
  "schemaVersion": "1.0",
  "items": [
    {
      "id": "edgeAgent",
      "filter": {
        "tail": 10
      }
    }
  ],
  "encoding": "none",
  "contentType": "text"
}
```

In the Azure portal, invoke the method with the method name `GetModuleLogs` and the following JSON payload:

JSON

```
{
    "schemaVersion": "1.0",
    "items": [
        {
            "id": "edgeAgent",
            "filter": {
                "tail": 10
            }
        }
    ],
    "encoding": "none",
    "contentType": "text"
}
```

The screenshot shows the Azure IoT Edge Device interface. A modal window titled 'Direct method' is open for the method 'GetMethodLogs'. The payload field contains the JSON code shown in the previous code block. The 'Result' field at the bottom displays the command-line output of the method execution.

```
[{"status":200,"payload":[{"id": "edgeAgent", "payloadBytes": null, "payload": "<6> 2020-09-14 19:02:25.984 +00:00 [INF] - Executing command: \\'Stop module edgeHub\\'\\n<6> 2020-09-14 19:02:25.986 +00:00 [INF] - Executing command: \\'Start module edgeHub\\'\\n<6> 2020-09-14 19:02:26.572 +00:00 [INF] - Executing command: \\'Saving edgeHub to store\\'\\n<6> 2020-09-14 19:02:26.573 +00:00 [INF] - Plan execution ended for deployment 3\\n<6> 2020-09-14 19:02:26.740 +00:00 [INF] - Updated reported properties\\n<6> 2020-09-14 19:04:32.683 +00:00 [INF] - Received direct method call - GetModuleLogs\\n<6> 2020-09-14 19:04:32.686 +00:00 [INF] - Received request GetModuleLogs with payload\\n<6> 2020-09-14 19:04:32.714 +00:00 [INF] - Processing request to get logs for [\\'schemaVersion\\':\\'1.0\\',\\'items\\':[\\{\\'id\\':\\'edgeAgent\\',\\'filter\\':\\'{\\'tail\\':10,\\'since\\':null,\\'until\\':null,\\'logLevel\\':\\'null\\',\\'regex\\':\\'null\\'}}],\\'encoding\\':\\'0\\',\\'contentType\\':\\'1\\'}]\\n<6> 2020-09-14 19:04:32.739 +00:00 [INF] - Received log options for edgeAgent with tail value 10\\n\\'"}]
```

You can also pipe the CLI output to Linux utilities, like [gzip](#), to process a compressed response. For example:

The screenshot shows the Azure CLI terminal. The command entered is:

```
az iot hub invoke-module-method \
--method-name 'GetModuleLogs' \
-n <hub name> \
-d <device id> \
-m '$edgeAgent' \
--method-payload '{"contentType": "text","schemaVersion": "1.0","encoding": "gzip","items": [{"id": "edgeHub","filter": {"since": "2d","tail": 1000}}]},' \
-o tsv --query 'payload[0].payloadBytes' \
| base64 --decode \
| gzip -d
```

Upload module logs

Use the **UploadModuleLogs** direct method to send the requested logs to a specified Azure Blob Storage container.

ⓘ Note

Use the `since` and `until` filter options to limit the range of logs retrieved. Calling this direct method without bounds retrieves all the logs which may be large, time consuming, or costly.

If you wish to upload logs from a device behind a gateway device, you will need to have the [API proxy and blob storage modules](#) configured on the top layer device. These modules route the logs from your lower layer device through your gateway device to your storage in the cloud.

This method accepts a JSON payload similar to **GetModuleLogs**, with the addition of the "sasUrl" key:

JSON

```
{  
    "schemaVersion": "1.0",  
    "sasUrl": "Full path to SAS URL",  
    "items": [  
        {  
            "id": "regex string",  
            "filter": {  
                "tail": "int",  
                "since": "string",  
                "until": "string",  
                "loglevel": "int",  
                "regex": "regex string"  
            }  
        }  
    ],  
    "encoding": "gzip/none",  
    "contentType": "json/text"  
}
```

[+] Expand table

Name	Type	Description
sasURL	string (URI)	Shared Access Signature URL with write access to Azure Blob Storage container.

A successful request to upload logs returns a "status": 200 followed by a payload with the following schema:

JSON
{ "status": "string", "message": "string", "correlationId": "GUID" }

[Expand table](#)

Name	Type	Description
status	string	One of NotStarted, Running, Completed, Failed, or Unknown.
message	string	Message if error, empty string otherwise.
correlationId	string	ID to query to status of the upload request.

For example:

The following invocation uploads the last 100 log lines from all modules, in compressed JSON format:

Azure CLI
<pre>az iot hub invoke-module-method --method-name UploadModuleLogs -n <hub name> -d <device id> -m '\$edgeAgent' --method-payload \ ' { "schemaVersion": "1.0", "sasUrl": "<sasUrl>", "items": [{ "id": ".*", "filter": { "tail": 100 } }], "encoding": "gzip", "contentType": "json" }</pre>

The following invocation uploads the last 100 log lines from edgeAgent and edgeHub with the last 1000 log lines from tempSensor module in uncompressed text format:

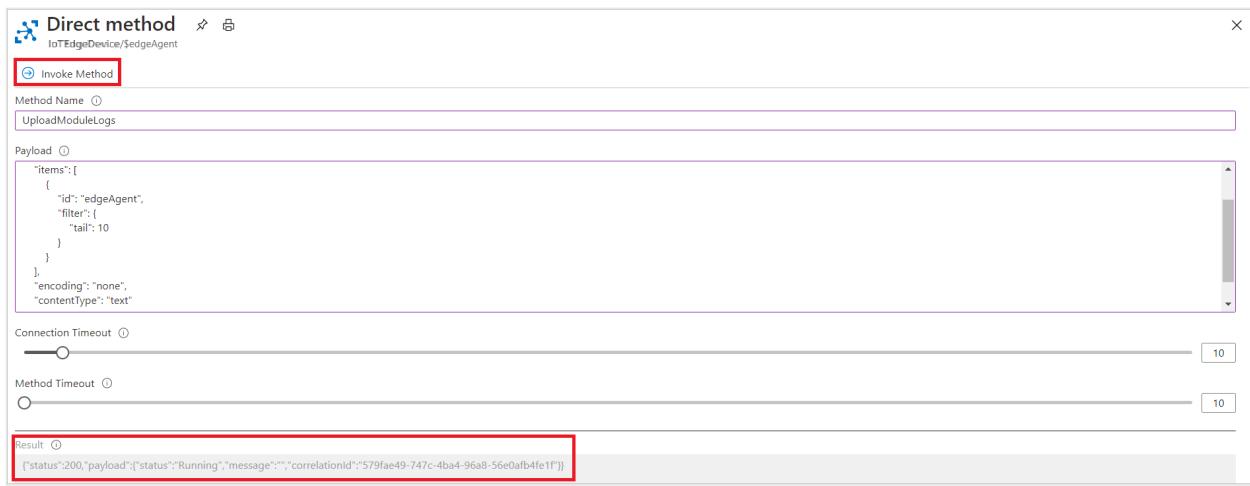
Azure CLI

```
az iot hub invoke-module-method --method-name UploadModuleLogs -n <hub name>
-d <device id> -m '$edgeAgent' --method-payload \
'
{
    "schemaVersion": "1.0",
    "sasUrl": "<sasUrl>",
    "items": [
        {
            "id": "edge",
            "filter": {
                "tail": 100
            }
        },
        {
            "id": "tempSensor",
            "filter": {
                "tail": 1000
            }
        }
    ],
    "encoding": "none",
    "contentType": "text"
}
'
```

In the Azure portal, invoke the method with the method name `UploadModuleLogs` and the following JSON payload after populating the sasURL with your information:

JSON

```
{
    "schemaVersion": "1.0",
    "sasUrl": "<sasUrl>",
    "items": [
        {
            "id": "edgeAgent",
            "filter": {
                "tail": 10
            }
        }
    ],
    "encoding": "none",
    "contentType": "text"
}
```



Upload support bundle diagnostics

Use the **UploadSupportBundle** direct method to bundle and upload a zip file of IoT Edge module logs to an available Azure Blob Storage container. This direct method runs the [iotedge support-bundle](#) command on your IoT Edge device to obtain the logs.

ⓘ Note

If you wish to upload logs from a device behind a gateway device, you will need to have the [API proxy and blob storage modules](#) configured on the top layer device. These modules route the logs from your lower layer device through your gateway device to your storage in the cloud.

This method accepts a JSON payload with the following schema:

JSON

```
{  
    "schemaVersion": "1.0",  
    "sasUrl": "Full path to SAS url",  
    "since": "2d",  
    "until": "1d",  
    "edgeRuntimeOnly": false  
}
```

[+] Expand table

Name	Type	Description
schemaVersion	string	Set to 1.0

Name	Type	Description
sasURL	string (URI)	Shared Access Signature URL with write access to Azure Blob Storage container
since	string	Only return logs since this time, as an rfc3339 timestamp, UNIX timestamp, or a duration (days (d) hours (h) minutes (m)). For example, a duration for one day, 12 hours, and 30 minutes can be specified as <i>1 day 12 hours 30 minutes</i> or <i>1d 12h 30m</i> . OPTIONAL.
until	string	Only return logs before the specified time, as an rfc3339 timestamp, UNIX timestamp, or duration (days (d) hours (h) minutes (m)). For example, a duration 90 minutes can be specified as <i>90 minutes</i> or <i>90m</i> . OPTIONAL.
edgeRuntimeOnly	boolean	If true, only return logs from Edge Agent, Edge Hub, and the Edge Security Daemon. Default: false. OPTIONAL.

ⓘ Important

IoT Edge support bundle may contain Personally Identifiable Information.

A successful request to upload logs returns a "status": 200 followed by a payload with the same schema as the [UploadModuleLogs](#) response:

JSON

```
{
  "status": "string",
  "message": "string",
  "correlationId": "GUID"
}
```

[+] [Expand table](#)

Name	Type	Description
status	string	One of <code>NotStarted</code> , <code>Running</code> , <code>Completed</code> , <code>Failed</code> , or <code>Unknown</code> .
message	string	Message if error, empty string otherwise.
correlationId	string	ID to query to status of the upload request.

For example:

Azure CLI

```

az iot hub invoke-module-method --method-name 'UploadSupportBundle' -n <hub
name> -d <device id> -m '$edgeAgent' --method-payload \
'

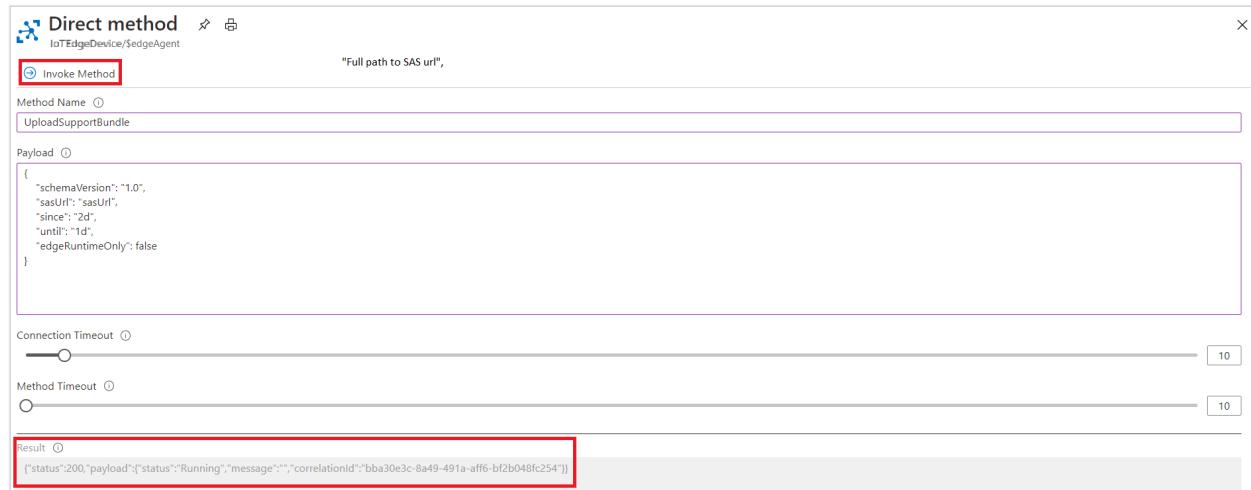
{
    "schemaVersion": "1.0",
    "sasUrl": "Full path to SAS url",
    "since": "2d",
    "until": "1d",
    "edgeRuntimeOnly": false
}

```

In the Azure portal, invoke the method with the method name `UploadSupportBundle` and the following JSON payload after populating the `sasURL` with your information:

JSON

```
{
    "schemaVersion": "1.0",
    "sasUrl": "Full path to SAS url",
    "since": "2d",
    "until": "1d",
    "edgeRuntimeOnly": false
}
```



Get upload request status

Use the `GetTaskStatus` direct method to query the status of an upload logs request. The `GetTaskStatus` request payload uses the `correlationId` of the upload logs request to get the task's status. The `correlationId` is returned in response to the `UploadModuleLogs` direct method call.

This method accepts a JSON payload with the following schema:

JSON

```
{  
    "schemaVersion": "1.0",  
    "correlationId": "<GUID>"  
}
```

A successful request to upload logs returns a "status": 200 followed by a payload with the same schema as the [UploadModuleLogs](#) response:

JSON

```
{  
    "status": "string",  
    "message": "string",  
    "correlationId": "GUID"  
}
```

[+] [Expand table](#)

Name	Type	Description
status	string	One of <code>NotStarted</code> , <code>Running</code> , <code>Completed</code> , <code>Failed</code> , ' <code>Cancelled</code> ', or <code>Unknown</code> .
message	string	Message if error, empty string otherwise.
correlationId	string	ID to query to status of the upload request.

For example:

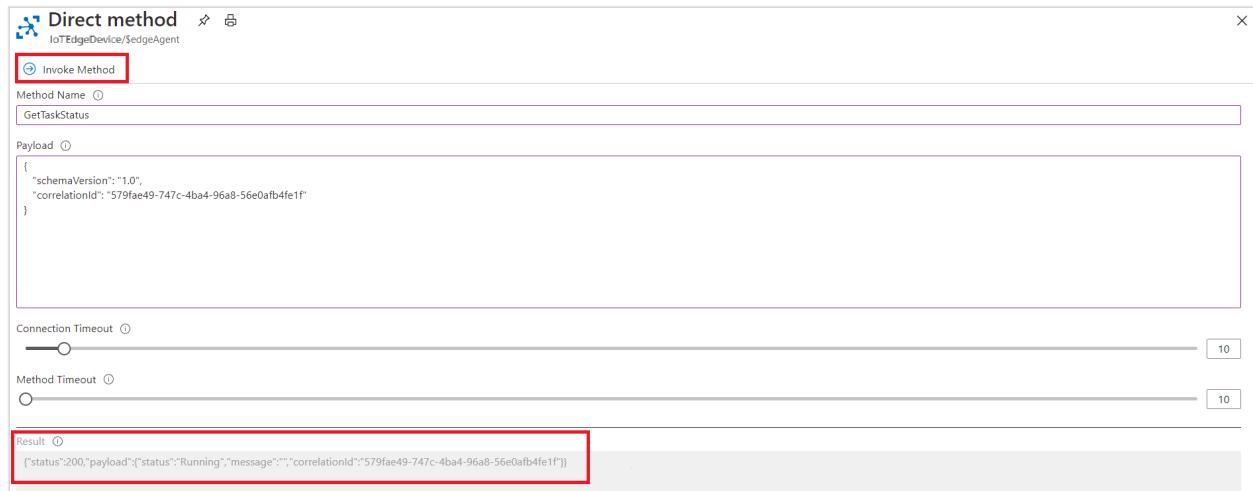
Azure CLI

```
az iot hub invoke-module-method --method-name 'GetTaskStatus' -n <hub name>  
-d <device id> -m '$edgeAgent' --method-payload \  
'  
{  
    "schemaVersion": "1.0",  
    "correlationId": "<GUID>"  
}'
```

In the Azure portal, invoke the method with the method name `GetTaskStatus` and the following JSON payload after populating the GUID with your information:

JSON

```
{  
    "schemaVersion": "1.0",  
    "correlationId": "<GUID>"  
}
```



Next steps

Properties of the IoT Edge agent and IoT Edge hub module twins

Access built-in metrics in Azure IoT Edge

Article • 04/09/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The IoT Edge runtime components, IoT Edge hub, and IoT Edge agent, produce built-in metrics in the [Prometheus exposition format](#). Access these metrics remotely to monitor and understand the health of an IoT Edge device.

You can use your own solution to access these metrics. Or, you can use the [metrics-collector module](#), which handles collecting the built-in metrics and sending them to Azure Monitor or Azure IoT Hub. For more information, see [Collect and transport metrics](#).

Metrics are automatically exposed by default on **port 9600** of the **edgeHub** and **edgeAgent** modules (`http://edgeHub:9600/metrics` and `http://edgeAgent:9600/metrics`). They aren't port mapped to the host by default.

Access metrics from the host by exposing and mapping the metrics port from the module's `createOptions`. The example below maps the default metrics port to port 9601 on the host:

JSON

```
{
  "ExposedPorts": {
    "9600/tcp": {}
  },
  "HostConfig": {
    "PortBindings": {
      "9600/tcp": [
        {
          "HostPort": "9601"
        }
      ]
    }
  }
}
```

Choose different and unique host port numbers if you are mapping both the `edgeHub` and `edgeAgent`'s metrics endpoints.

Note

The environment variable `httpSettings__enabled` should not be set to `false` for built-in metrics to be available for collection.

Environment variables that can be used to disable metrics are listed in the [azure/iotedge repo doc ↗](#).

Available metrics

Metrics contain tags to help identify the nature of the metric being collected. All metrics contain the following tags:

 Expand table

Tag	Description
iohub	The hub the device is talking to
edge_device	The ID of the current device
instance_number	A GUID representing the current runtime. On restart, all metrics are reset. This GUID makes it easier to reconcile restarts.

In the Prometheus exposition format, there are four core metric types: counter, gauge, histogram, and summary. For more information about the different metric types, see the [Prometheus metric types documentation ↗](#).

The quantiles provided for the built-in histogram and summary metrics are 0.1, 0.5, 0.9 and 0.99.

The **edgeHub** module produces the following metrics:

 Expand table

Name	Dimensions	Description
<code>edgehub_gettwin_total</code>	<code>source</code> (operation source) <code>id</code> (module ID)	Type: counter Total number of GetTwin calls
<code>edgehub_messages_received_total</code>	<code>route_output</code> (output that sent message) <code>id</code>	Type: counter Total number of messages received from clients

Name	Dimensions	Description
edgehub_messages_sent_total	<code>from</code> (message source) <code>to</code> (message destination) <code>from_route_output</code> <code>to_route_input</code> (message destination input) <code>priority</code> (message priority to destination)	Type: counter Total number of messages sent to clients or upstream <code>to_route_input</code> is empty when <code>to</code> is \$upstream
edgehub_reported_properties_total	<code>target</code> (update target) <code>id</code>	Type: counter Total reported property updates calls
edgehub_message_size_bytes	<code>id</code>	Type: summary Message size from clients Values may be reported as <code>Nan</code> if no new measurements are reported for a certain period of time (currently 10 minutes); for <code>summary</code> type, corresponding <code>_count</code> and <code>_sum</code> counters are emitted.
edgehub_gettwin_duration_seconds	<code>source</code> <code>id</code>	Type: summary Time taken for get twin operations
edgehub_message_send_duration_seconds	<code>from</code> <code>to</code> <code>from_route_output</code> <code>to_route_input</code>	Type: summary Time taken to send a message
edgehub_message_process_duration_seconds	<code>from</code> <code>to</code> <code>priority</code>	Type: summary Time taken to process a message from the queue
edgehub_reported_properties_update_duration_seconds	<code>target</code> <code>id</code>	Type: summary Time taken to update reported properties

Name	Dimensions	Description
edgehub_direct_method_duration_seconds	from (caller) to (receiver)	Type: summary Time taken to resolve a direct message
edgehub_direct_methods_total	from to	Type: counter Total number of direct messages sent
edgehub_queue_length	endpoint (message source) priority (queue priority)	Type: gauge Current length of edgeHub's queue for a given priority
edgehub_messages_dropped_total	reason (no_route, ttl_expiry) from from_route_output	Type: counter Total number of messages removed because of reason
edgehub_messages_unack_total	reason (storage_failure) from from_route_output	Type: counter Total number of messages unacknowledged because storage failure
edgehub_offline_count_total	id	Type: counter Total number of times edgeHub went offline
edgehub_offline_duration_seconds	id	Type: summary Time edge hub was offline
edgehub_operation_retry_total	id operation (operation name)	Type: counter Total number of times edgeHub operations were retried
edgehub_client_connect_failed_total	id reason (not authenticated)	Type: counter Total number of times clients failed to connect to edgeHub

The **edgeAgent** module produces the following metrics:

[Expand table](#)

Name	Dimensions	Description
edgeAgent_total_time_running_correctly_seconds	module_name	Type: gauge The amount of time the module was specified in the deployment and was in the running state
edgeAgent_total_time_expected_running_seconds	module_name	Type: gauge The amount of time the module was specified in the deployment
edgeAgent_module_start_total	module_name, module_version	Type: counter Number of times edgeAgent asked docker to start the module
edgeAgent_module_stop_total	module_name, module_version	Type: counter Number of times edgeAgent asked docker to stop the module
edgeAgent_command_latency_seconds	command	Type: gauge How long it took docker to execute the given command. Possible commands are: create, update, remove, start, stop, and restart
edgeAgent_iothub_syncs_total		Type: counter Number of times edgeAgent attempted to sync its twin with ioHub, both successful and unsuccessful. This number includes both Agent requesting a twin and Hub notifying of a twin update
edgeAgent_unsuccessful_iothub_syncs_total		Type: counter Number of times edgeAgent failed to sync its twin with ioHub.
edgeAgent_deployment_time_seconds		Type: counter The amount of time it took to complete a new deployment after receiving a change.
edgeagent_direct_method_invocations_count	method_name	Type: counter Number of times a built-in

Name	Dimensions	Description
		edgeAgent direct method is called, such as Ping or Restart.
edgeAgent_host_uptime_seconds		Type: gauge How long the host has been on
edgeAgent_iotedged_uptime_seconds		Type: gauge How long iotedged has been running
edgeAgent_available_disk_space_bytes	disk_name, disk_filesystem, disk_filetype	Type: gauge Amount of space left on the disk
edgeAgent_total_disk_space_bytes	disk_name, disk_filesystem, disk_filetype	Type: gauge Size of the disk
edgeAgent_used_memory_bytes	module_name	Type: gauge Amount of RAM used by all processes
edgeAgent_total_memory_bytes	module_name	Type: gauge RAM available
edgeAgent_used_cpu_percent	module_name	Type: histogram Percent of cpu used by all processes
edgeAgent_created_pids_total	module_name	Type: gauge The number of processes or threads the container has created
edgeAgent_total_network_in_bytes	module_name	Type: gauge The number of bytes received from the network
edgeAgent_total_network_out_bytes	module_name	Type: gauge The number of bytes sent to network
edgeAgent_total_disk_read_bytes	module_name	Type: gauge The number of bytes read from the disk
edgeAgent_total_disk_write_bytes	module_name	Type: gauge The number of bytes written to disk

Name	Dimensions	Description
edgeAgent_metadata	edge_agent_version, experimental_features, host_information	Type: gauge General metadata about the device. The value is always 0, information is encoded in the tags. Note <code>experimental_features</code> and <code>host_information</code> are json objects. <code>host_information</code> looks like <pre>{"OperatingSystemType": "linux", "Architecture": "x86_64", "Version": "1.2.7", "Provisioning": {"Type": "dps.tpm", "DynamicReprovisioning": false, "AlwaysReprovisionOnStartup": false}, "ServerVersion": "20.10.11+azure-3", "KernelVersion": "5.11.0-1027-azure", "OperatingSystem": "Ubuntu 20.04.4 LTS", "NumCpus": 2, "Virtualized": "yes"}</pre> . Note <code>ServerVersion</code> is the Docker version and <code>Version</code> is the IoT Edge security daemon version.

Next steps

- Collect and transport metrics
- Understand the Azure IoT Edge runtime and its architecture
- Properties of the IoT Edge agent and IoT Edge hub module twins

Collect and transport metrics

Article • 06/10/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

You can remotely monitor your IoT Edge fleet using Azure Monitor and built-in metrics integration. To enable this capability on your device, add the metrics-collector module to your deployment and configure it to collect and transport module metrics to Azure Monitor.

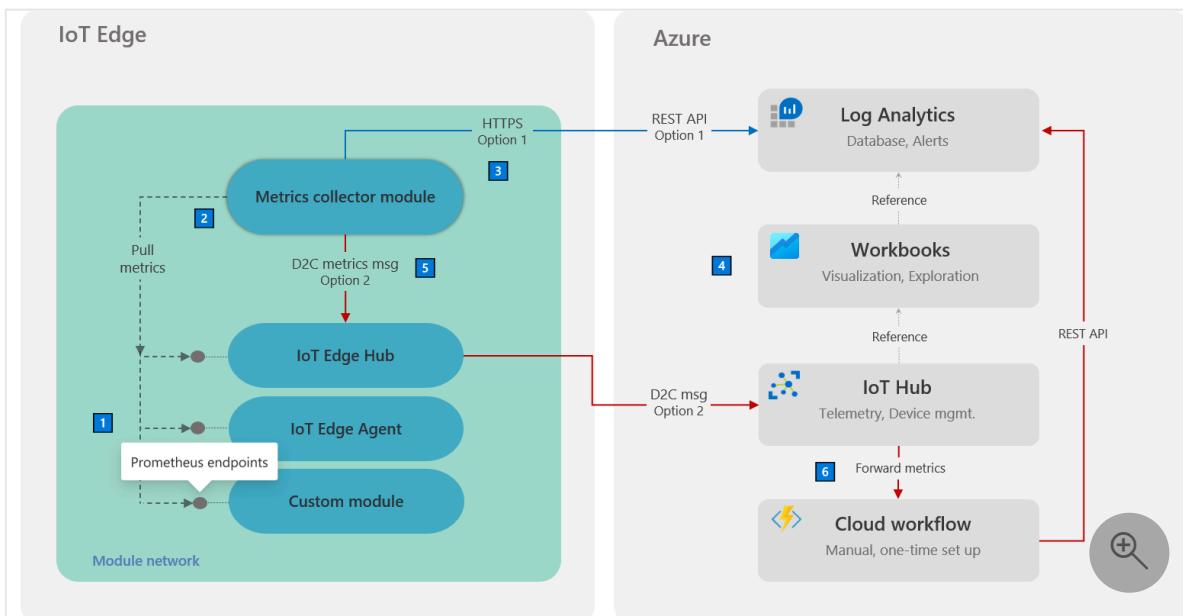
To configure monitoring on your IoT Edge device, follow the [Tutorial: Monitor IoT Edge devices](#). You learn how to add the metrics-collector module to your device. This article gives you an overview of the monitoring architecture and explains your options on configuring metrics on your device.

<https://aka.ms/docs/player?id=94a7d988-4a35-4590-9dd8-a511cdd68bee>

[IoT Edge integration with Azure Monitor](#)  (4:06)

Architecture





[Expand table](#)

Note	Description
1	All modules must emit metrics using the Prometheus data model . While built-in metrics enable broad workload visibility by default, custom modules can also be used to emit scenario-specific metrics to enhance the monitoring solution. Learn how to instrument custom modules using open-source libraries in the Add custom metrics article.
2	The metrics-collector module is a Microsoft-supplied IoT Edge module that collects workload module metrics and transports them off-device. Metrics collection uses a <i>pull</i> model. Collection frequency, endpoints, and filters can be configured to control the data egressed from the module. For more information, see metrics collector configuration section later in this article.
3	You have two options for sending metrics from the metrics-collector module to the cloud. <i>Option 1</i> sends the metrics to Log Analytics. ¹ The collected metrics are ingested into the specified Log Analytics workspace using a fixed, native table called <code>InsightsMetrics</code> . This table's schema is compatible with the Prometheus metrics data model.
	This option requires access to the workspace on outbound port 443. The Log Analytics workspace ID and key must be specified as part of the module configuration. To enable in restricted networks, see Enable in restricted network access scenarios later in this article.
4	Each metric entry contains the <code>ResourceId</code> that was specified as part of module configuration . This association automatically links the metric with the specified resource (for example, IoT Hub). As a result, the curated IoT Edge workbook templates can retrieve metrics by issuing queries against the resource.

Note	Description
	This approach also allows multiple IoT hubs to safely share a single Log Analytics workspace as a metrics database.
5	<i>Option 2</i> sends the metrics to IoT Hub. ¹ The collector module can be configured to send the collected metrics as UTF-8 encoded JSON device-to-cloud messages via the <code>edgeHub</code> module. This option unlocks monitoring of locked-down IoT Edge devices that are allowed external access to only the IoT Hub endpoint. It also enables monitoring of child IoT Edge devices in a nested configuration where child devices can only access their parent device.
6	When metrics are routed via IoT Hub, a (one-time) cloud workflow needs to be set up. The workflow processes messages arriving from the metrics-collector module and sends them to the Log Analytics workspace. The workflow enables the curated visualizations and alerts functionality even for metrics arriving via this optional path. See the Route metrics through IoT Hub section for details on how to set up this cloud workflow.

¹ Currently, using *option 1* to directly transport metrics to Log Analytics from the IoT Edge device is the simpler path that requires minimal setup. The first option is preferred unless your specific scenario demands the *option 2* approach so that the IoT Edge device communicates only with IoT Hub.

Metrics collector module

A Microsoft-supplied metrics-collector module can be added to an IoT Edge deployment to collect module metrics and send them to Azure Monitor. The module code is open-source and available in the [IoT Edge GitHub repo](#).

The metrics-collector module is provided as a multi-arch Docker container image that supports Linux X64, ARM32, ARM64, and Windows X64 (version 1809). It's publicly available at mcr.microsoft.com/azureiotedge-metrics-collector.

Metrics collector configuration

All configuration for the metrics-collector is done using environment variables. Minimally, the variables noted in this table marked as **Required** need to be specified.

IoT Hub	Expand table
---------	------------------------------

Environment variable	Description
<code>name</code>	
<code>ResourceId</code>	<p>Resource ID of the IoT hub that the device communicates with. For more information, see the Resource ID section.</p>
	Required
	Default value: <i>none</i>
<code>UploadTarget</code>	<p>Controls whether metrics are sent directly to Azure Monitor over HTTPS or to IoT Hub as D2C messages. For more information, see upload target.</p>
	Can be either <code>AzureMonitor</code> or <code>IoTMessage</code>
	Not required
	Default value: <code>AzureMonitor</code>
<code>LogAnalyticsWorkspaceId</code>	<p>Log Analytics workspace ID.</p>
	Required only if <code>UploadTarget</code> is <code>AzureMonitor</code>
	Default value: <i>none</i>
<code>LogAnalyticsSharedKey</code>	<p>Log Analytics workspace key.</p>
	Required only if <code>UploadTarget</code> is <code>AzureMonitor</code>
	Default value: <i>none</i>
<code>ScrapeFrequencyInSecs</code>	<p>Recurring time interval in seconds at which to collect and transport metrics.</p>
	Example: <i>600</i>
	Not required
	Default value: <i>300</i>
<code>MetricsEndpointsCSV</code>	<p>Comma-separated list of endpoints to collect Prometheus metrics from. All module endpoints to collect metrics from must appear in this list.</p>
	<p>Example: <i>http://edgeAgent:9600/metrics, http://edgeHub:9600/metrics, http://MetricsSpewer:9417/metrics</i></p>
	Not required

Environment variable name	Description
	Default value: <i>http://edgeHub:9600/metrics, http://edgeAgent:9600/metrics</i>
<code>AllowedMetrics</code>	List of metrics to collect, all other metrics are ignored. Set to an empty string to disable. For more information, see allow and disallow lists .
	Example: <i>metricToScrape{quantile=0.99} [endpoint=http://MetricsSpewer:9417/metrics]</i>
	Not required
	Default value: <i>empty</i>
<code>BlockedMetrics</code>	List of metrics to ignore. Overrides <code>AllowedMetrics</code> , so a metric isn't reported if it's included in both lists. For more information, see allow and disallow lists .
	Example: <i>metricToIgnore{quantile=0.5} [endpoint=http://VeryNoisyModule:9001/metrics], docker_container_disk_write_bytes</i>
	Not required
	Default value: <i>empty</i>
<code>CompressForUpload</code>	Controls if compression should be used when uploading metrics. Applies to all upload targets.
	Example: <i>true</i>
	Not required
	Default value: <i>true</i>
<code>AzureDomain</code>	Specifies the top-level Azure domain to use when ingesting metrics directly to Log Analytics.
	Example: <i>azure.us</i>
	Not required
	Default value: <i>azure.com</i>

Resource ID

IoT Hub

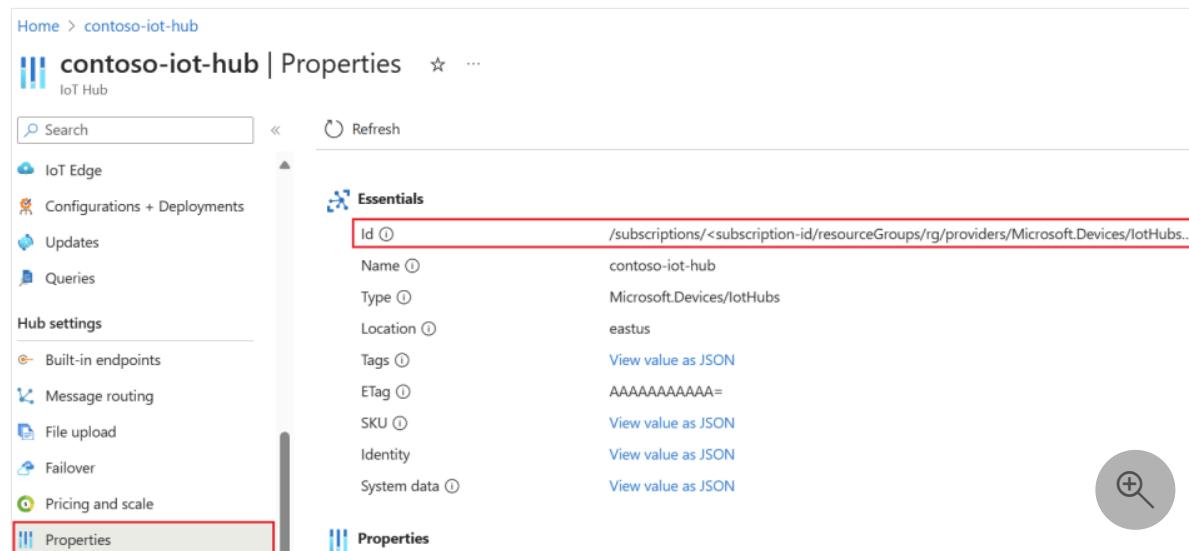
The metrics-collector module requires the Azure Resource Manager ID of the IoT hub that the IoT Edge device belongs to. Provide this ID as the value of the **ResourceID** environment variable.

The resource ID takes the following format:

input

```
/subscriptions/<subscription id>/resourceGroups/<resource group name>/providers/Microsoft.Devices/IoTHubs/<iot hub name>
```

You can find the resource ID in the **Properties** page of the IoT hub in the Azure portal.



Id	/subscriptions/<subscription-id>/resourceGroups/rg/providers/Microsoft.Devices/IoTHubs/<iot hub name>
Name	contoso-iot-hub
Type	Microsoft.Devices/IoTHubs
Location	eastus
Tags	View value as JSON
Etag	AAAAAAAAAA=
SKU	View value as JSON
Identity	View value as JSON
System data	View value as JSON

Or, you retrieve the ID with the [az resource show](#) command:

Azure CLI

```
az resource show -g <resource group> -n <hub name> --resource-type "Microsoft.Devices/IoTHubs"
```

Upload target

IoT Hub

The **UploadTarget** configuration option controls whether metrics are sent directly to Azure Monitor or to IoT Hub.

If you set `UploadTarget` to `IoTMessage`, then your module metrics are published as IoT messages. These messages are emitted as UTF8-encoded json from the endpoint `/messages/modules/<metrics collector module name>/outputs/metricOutput`. For example, if your IoT Edge Metrics Collector module is named `IoTEdgeMetricsCollector`, the endpoint is `/messages/modules/IoTEdgeMetricsCollector/outputs/metricOutput`. The format is as follows:

JSON

```
[{  
    "TimeGeneratedUtc": "<time generated>",  
    "Name": "<prometheus metric name>",  
    "Value": <decimal value>,  
    "Label": {  
        "<label name>": "<label value>"  
    }  
, {  
    "TimeGeneratedUtc": "2020-07-28T20:00:43.2770247Z",  
    "Name": "docker_container_disk_write_bytes",  
    "Value": 0.0,  
    "Label": {  
        "name": "AzureMonitorForIotEdgeModule"  
    }  
}]
```

Allow and disallow lists

The `AllowedMetrics` and `BlockedMetrics` configuration options take space- or comma-separated lists of metric selectors. A metric matches the list and is included or excluded if it matches one or more metrics in either list.

Metric selectors use a format similar to a subset of the [PromQL](#) query language.

query

```
metricToSelect{quantile=0.5,otherLabel=~Re[ge]*|x}  
[http://VeryNoisyModule:9001/metrics]
```

Metric selectors consist of three parts:

Metric name (`metricToSelect`).

- Wildcards `*` (any characters) and `?` (any single character) can be used in metric names. For example, `*CPU` would match `maxCPU` and `minCPU` but not `CPUMaximum`.

`???CPU` would match `maxCPU` and `minCPU` but not `maximumCPU`.

- This component is required in a metrics selector.

Label-based selectors (`{quantile=0.5,otherLabel=~Re[ge]*|x}`).

- Multiple metric values can be included in the curly brackets. The values should be comma-separated.
- A metric is matched if at least all labels in the selector are present and also match.
- Like PromQL, the following matching operators are allowed.
 - `=` Match labels exactly equal to the provided string (case sensitive).
 - `!=` Match labels not exactly equal to the provided string.
 - `=~` Match labels to a provided regex. ex: `label=~CPU|Mem|[0-9]*`
 - `!~` Match labels that don't fit a provided regex.
 - Regex is fully anchored (A `^` and `$` are automatically added to the start and end of each regex)
 - This component is optional in a metrics selector.

Endpoint selector (`[http://VeryNoisyModule:9001/metrics]`).

- The URL should exactly match a URL listed in `MetricsEndpointsCSV`.
- This component is optional in a metrics selector.

A metric must match all parts of a given selector to be selected. It must match the name *and* have all the same labels with matching values *and* come from the given endpoint.

For example, `mem{quantile=0.5,otherLabel=foobar}`

`[http://VeryNoisyModule:9001/metrics]` wouldn't match the selector

`mem{quantile=0.5,otherLabel=~foo|bar}[http://VeryNoisyModule:9001/metrics]`.

Multiple selectors should be used to create or-like behavior instead of and-like behavior.

For example, to allow the custom metric `mem` with any label from a module `module1` but only allow the same metric from `module2` with the label `agg=p99`, the following selector can be added to `AllowedMetrics`:

query

```
mem{}[http://module1:9001/metrics] mem{agg="p99"}  
[http://module2:9001/metrics]
```

Or, to allow the custom metrics `mem` and `cpu` for any labels or endpoint, add the following to `AllowedMetrics`:

```
query
```

```
mem cpu
```

Enable in restricted network access scenarios

If you're sending metrics directly to the Log Analytics workspace, allow outbound access to the following URLs:

- `https://<LOG_ANALYTICS_WORKSPACE_ID>.ods.opinsights.azure.com/*`
- `https://<LOG_ANALYTICS_WORKSPACE_ID>.oms.opinsights.azure.com/*`

Proxy considerations

The metrics-collector module is written in .NET Core. So use the same guidance as for system modules to [allow communication through a proxy server](#).

Metrics collection from local modules uses http protocol. Exclude local communication from going through the proxy server by setting the `NO_PROXY` environment variable.

Set `NO_PROXY` value to a comma-separated list of hostnames that should be excluded. Use module names for hostnames. For example: `edgeHub,edgeAgent,myCustomModule`.

Route metrics

IoT Hub

Sometimes it's necessary to ingest metrics through IoT Hub instead of sending them directly to Log Analytics. For example, when monitoring [IoT Edge devices in a nested configuration](#) where child devices have access only to the IoT Edge hub of their parent device. Another example is when deploying an IoT Edge device with *outbound network access only* to IoT Hub.

To enable monitoring in this scenario, the metrics-collector module can be configured to send metrics as device-to-cloud (D2C) messages via the `edgeHub` module. The capability can be turned on by setting the `uploadTarget` environment variable to `IoTMessage` in the collector [configuration](#).

 Tip

Remember to add an edgeHub route to deliver metrics messages from the collector module to IoT Hub. It looks like `FROM /messages/modules/replace-with-collector-module-name/* INTO $upstream.`

This option does require extra setup, a cloud workflow, to deliver metrics messages arriving at IoT Hub to the Log Analytics workspace. Without this set up, the other portions of the integration such as [curated visualizations](#) and [alerts](#) don't work.

 **Note**

Be aware of additional costs with this option. Metrics messages will count against your IoT Hub message quota. You will also be charged for Log Analytics ingestion and cloud workflow resources.

Sample cloud workflow

A cloud workflow that delivers metrics messages from IoT Hub to Log Analytics is available as part of the [IoT Edge logging and monitoring sample](#). The sample can be deployed on to existing cloud resources or serve as a production deployment reference.

Next steps

Explore the types of [curated visualizations](#) that Azure Monitor enables.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Explore curated visualizations in Azure IoT Edge

Article • 04/09/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

You can visually explore metrics collected from IoT Edge devices using Azure Monitor workbooks. Curated monitoring workbooks for IoT Edge devices are provided in the form of public templates:

- For devices connected to IoT Hub, from the [IoT Hub](#) page in the Azure portal, navigate to the [Workbooks](#) page in the [Monitoring](#) section.
- For devices connected to IoT Central, from the [IoT Central](#) page in the Azure portal, navigate to the [Workbooks](#) page in the [Monitoring](#) section.

Curated workbooks use [built-in metrics](#) from the IoT Edge runtime. They first need metrics to be [ingested](#) into a Log Analytics workspace. These views don't need any metrics instrumentation from the workload modules.

Access curated workbooks

Azure Monitor workbooks for IoT are a set of templates that you can use to visualize your device metrics. They can be customized to fit your solution.

To access the curated workbooks, use the following steps:

1. Sign in to the [Azure portal](#)  and navigate to your IoT Hub or IoT Central application.
2. Select [Workbooks](#) from the [Monitoring](#) section of the menu.
3. Choose the workbook that you want to explore from the list of public templates:
 - **Fleet View:** Monitor your fleet of devices across multiple IoT Hubs or Central Apps, and drill into specific devices for a health snapshot.

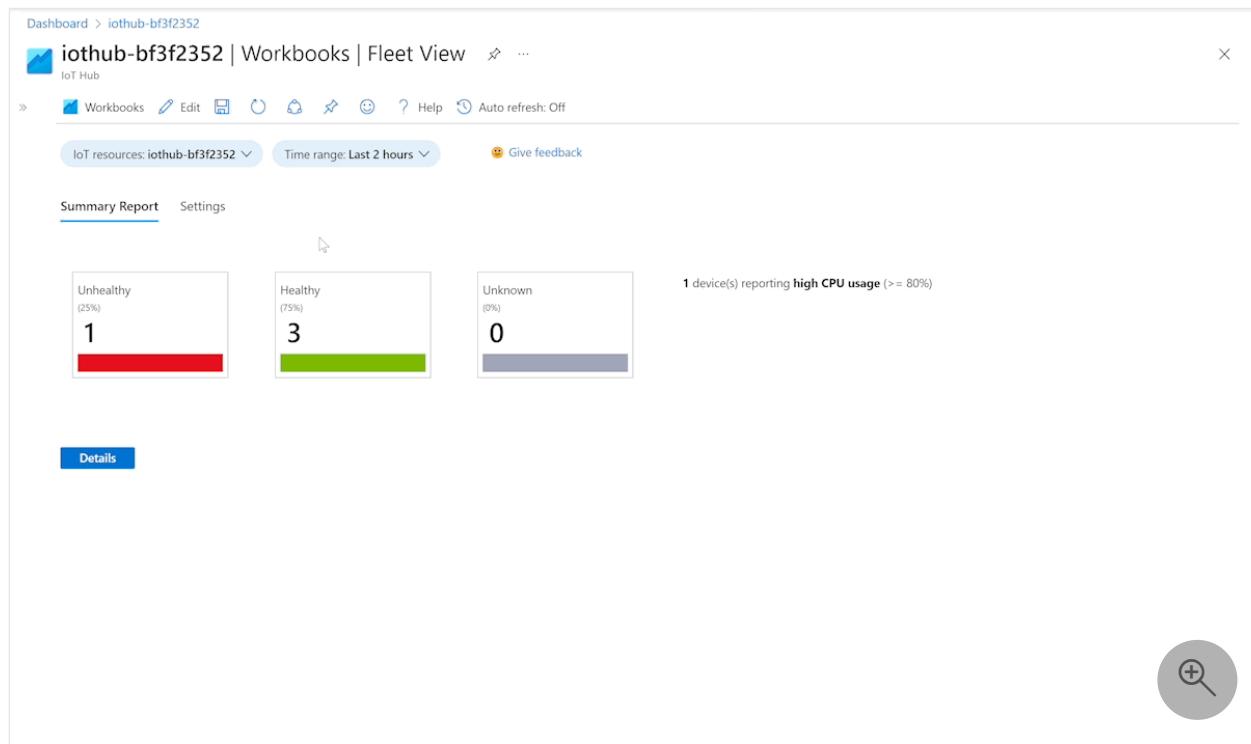
- **Device Details:** Visualize device details around messaging, modules, and host components on an IoT Edge device.
- **Alerts:** View triggered [alerts](#) for devices across multiple IoT resources.

Use the following sections to get a preview of the kind of data and visualizations that each workbook offers.

ⓘ Note

The screen captures that follow may not reflect the latest workbook design.

Fleet view workbook



By default, this view shows the health of devices associated with the current IoT cloud resource. You can select multiple IoT resources using the dropdown control on the top left.

Use the **Settings** tab to adjust the various thresholds to categorize the device as Healthy or Unhealthy.

Select the **Details** button to see the device list with a snapshot of aggregated, primary metrics. Select the link in the **Status** column to view the trend of an individual device's health metrics or the device name to view its detailed metrics.

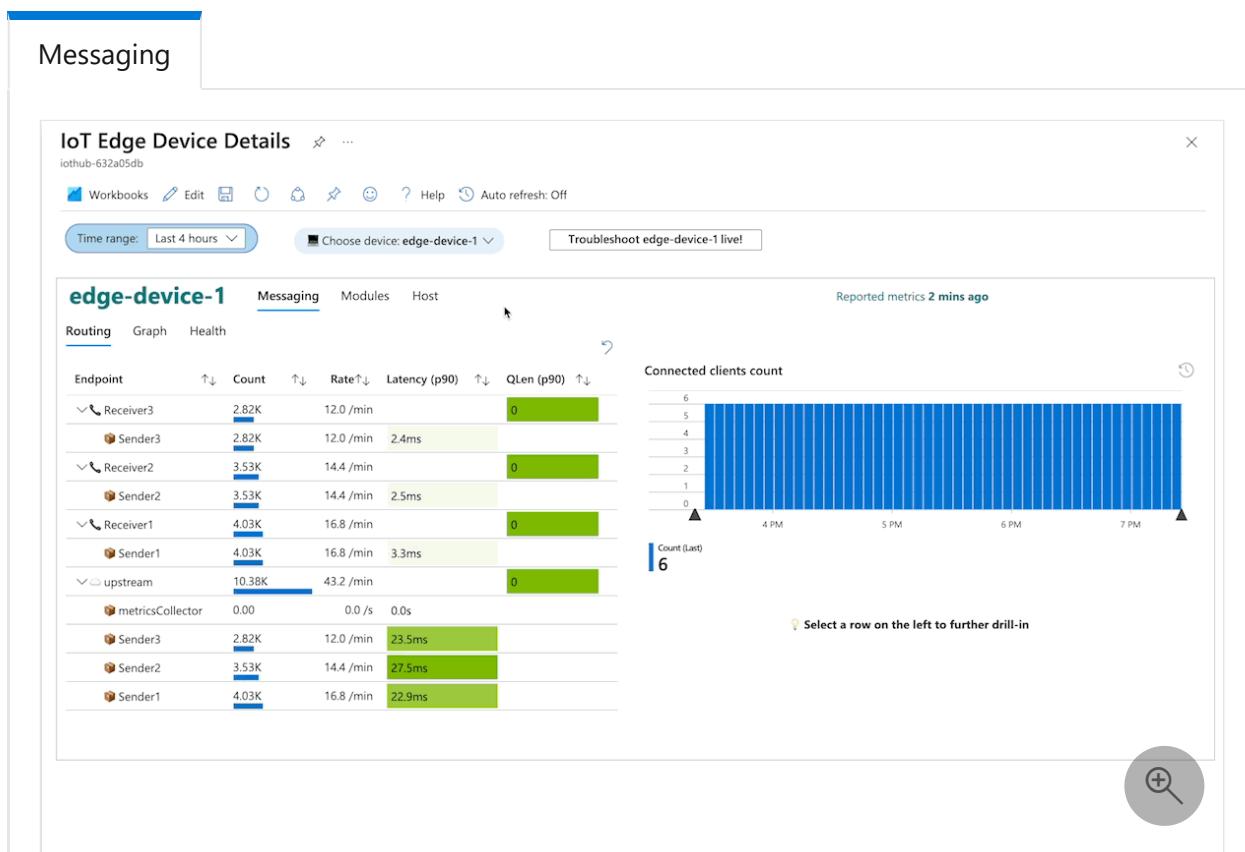
Device details workbook

The device details workbook has three views:

- The **Messaging** view visualizes the message routes for the device and reports on the overall health of the messaging system.
- The **Modules** view shows how the individual modules on a device are performing.
- The **Host** view shows information about the host device including version information for host components and resource use.

Switch between the views using the tabs at the top of the workbook.

The device details workbook also integrates with the IoT Edge portal-based troubleshooting experience. You can pull **Live logs** from your device using this feature. Access this experience by selecting the **Troubleshoot <device name> live** button above the workbook.



The **Messaging** view includes three subsections: routing details, a routing graph, and messaging health. Drag and let go on any time chart to adjust the global time range to the selected range.

The **Routing** section shows message flow between sending modules and receiving modules. It presents information such as message count, rate, and number of connected clients. Select a sender or receiver to drill in further. Clicking a sender shows the latency trend chart experienced by the sender and number of messages

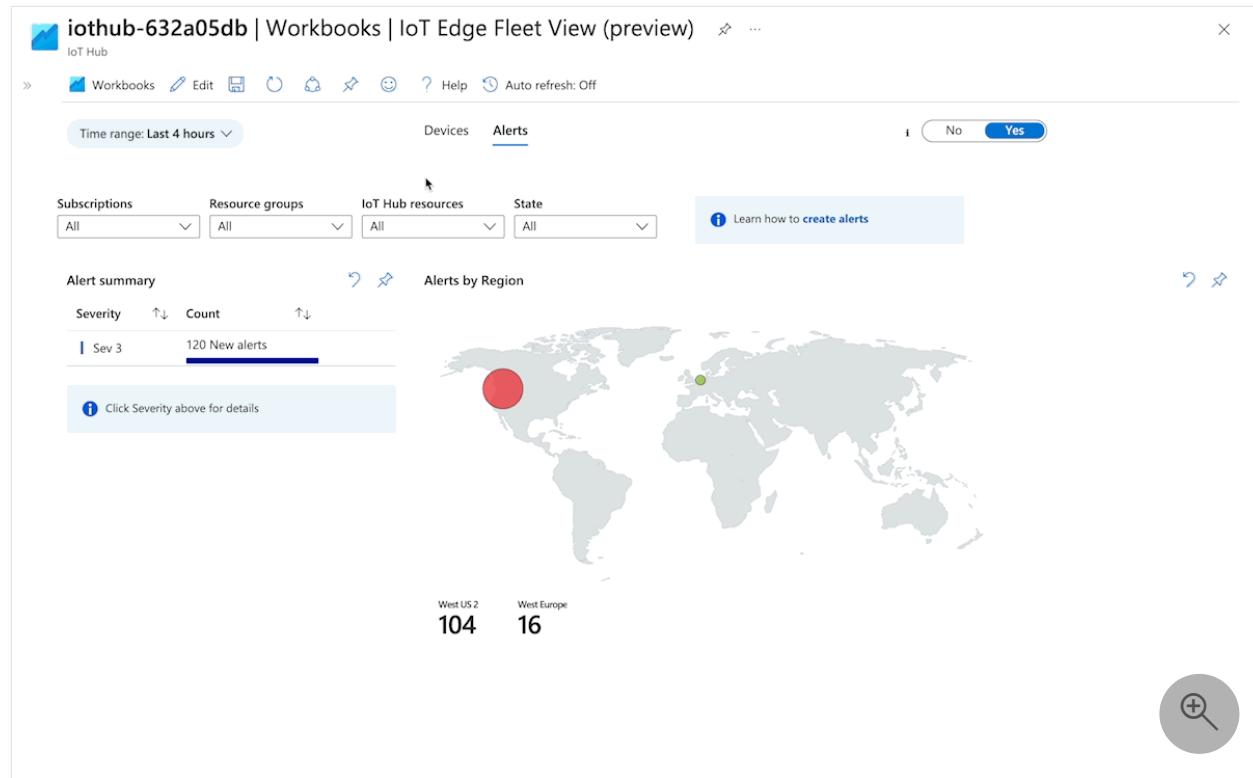
it sent. Clicking a receiver shows the queue length trend for the receiver and number of messages it received.

The **Graph** section shows a visual representation of message flow between modules. Drag and zoom to adjust the graph.

The **Health** section presents various metrics related to overall health of the messaging subsystem. Progressively drill-in to details if any errors are noted.

Alerts workbook

See the generated alerts from [pre-created alert rules](#) in the **Alerts** workbook. This view lets you see alerts from multiple IoT Hubs or IoT Central applications.



Select a severity row to see alerts details. The **Alert rule** link takes you to the alert context and the **Device** link opens the detailed metrics workbook. When opened from this view, the device details workbook is automatically adjusted to the time range around when the alert fired.

Customize workbooks

[Azure Monitor workbooks](#) are very customizable. You can edit the public templates to suit your requirements. All the visualizations are driven by resource-centric [Kusto Query Language](#) queries on the [InsightsMetrics](#) table.

To begin customizing a workbook, first enter editing mode. Select the **Edit** button in the menu bar of the workbook. Curated workbooks make extensive use of workbook groups. You may need to select **Edit** on several nested groups before being able to view a visualization query.

Save your changes as a new workbook. You can [share](#) the saved workbook with your team or [deploy them programmatically](#) as part of your organization's resource deployments.

Next steps

Customize your monitoring solution with [alert rules](#) and [metrics from custom modules](#).

Get notified about issues using alerts

Article • 12/15/2022

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Use [Azure Monitor Log alerts](#) to monitor IoT Edge devices at scale. As highlighted in the [solution architecture](#), Azure Monitor Log Analytics is used as the metrics database. This integration unlocks powerful and flexible alerting capabilities using resource-centric log alerts.

Important

This feature is currently only available for IoT Hub and not for IoT Central.

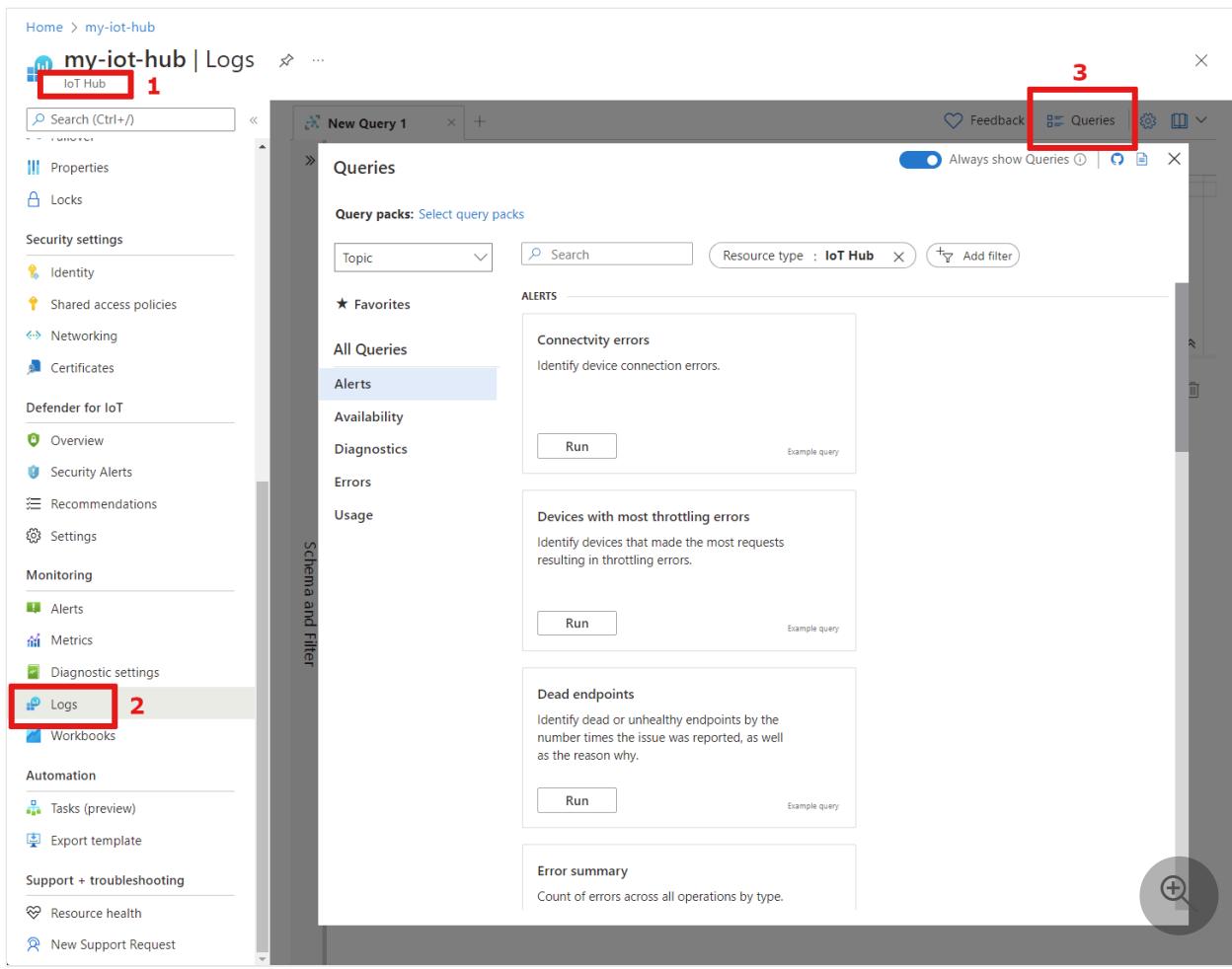
Create an alert rule

You can [create a log alert rule](#) for monitoring a broad range of conditions across your device fleet.

Sample [KQL](#) alert queries are provided under the IoT Hub resource. Queries that operate on metrics data from edge devices are prefixed with *IoT Edge*: in their title. Use these examples as-is or modify them as needed to create a query for your exact need.

To access the example alert queries, use the following steps:

1. Sign in to the [Azure portal](#)  and navigate to your IoT hub.
2. Select **Logs** from the **Monitoring** section of the menu.
3. The **Queries** example query browser will automatically open. If this is your first time to **Logs** you may have to close a video tutorial before you can see the query browser. The **Queries** tab can be used to bring up the example query browser again if you don't see it.



The [metrics-collector module](#) ingests all data into the standard [InsightsMetrics](#) table. You can create alert rules based on metrics data from custom modules by querying the same table.

Split by device dimension

All the example alert rule queries aggregate values by device ID. This grouping is needed to determine which device caused the alert to fire. You can select specific devices to enable the alert rule on or enable on all devices. Use the preview graph to explore the trend per device before setting the alert logic.

Choose notification preferences

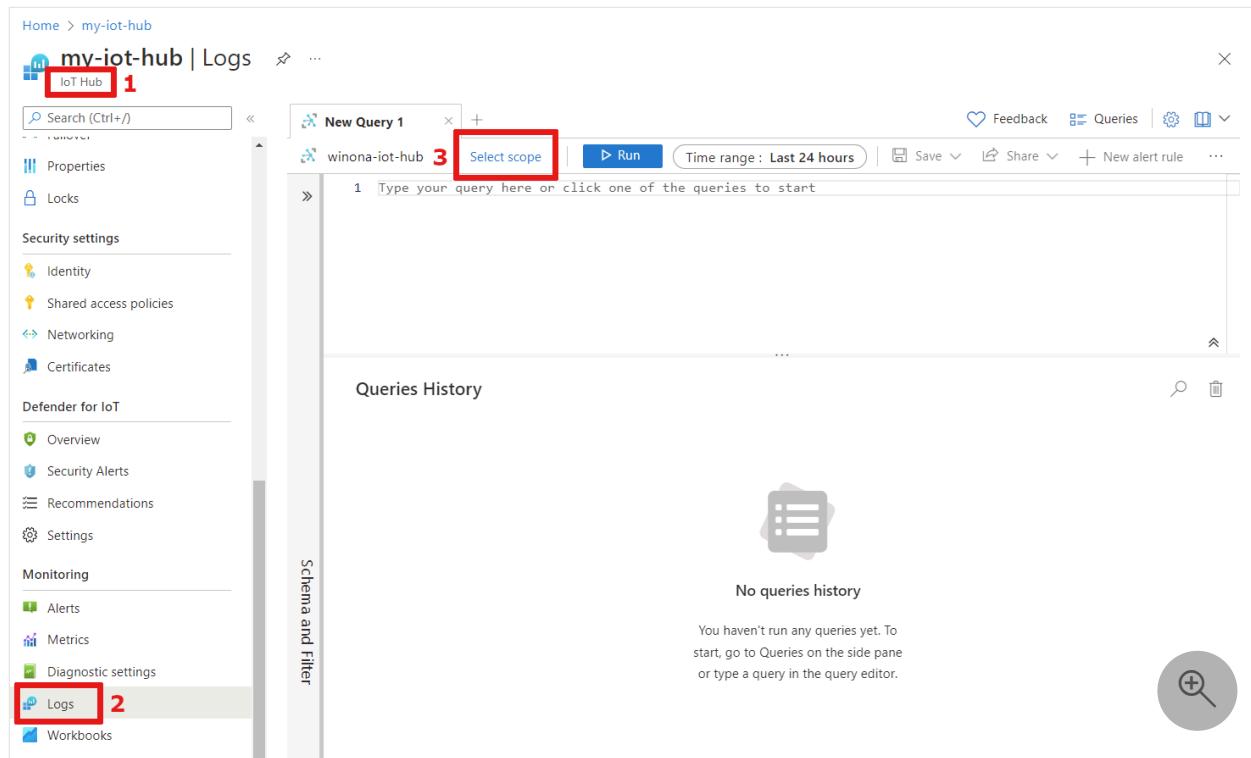
Configure your notification preferences in an [action group](#) and associate it with an alert rule when creating an alert rule.

Select alert rule scope

Using the guidance in the previous section creates an alert rule scoped to a single IoT hub. However, you might want to create the same rule for multiple IoT hubs. Change the

scope to a resource group or an entire subscription to enable the alert rule on all hubs within that scope.

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. Select **Logs** from the **Monitoring** section of the menu.
3. Select **Select scope** to change the scope of an alert rule.



Aggregate values by the `_ResourceId` field and choose it as the *Resource ID column* when creating the alert rule. This approach will associate an alert with the correct resource for convenience.

Viewing alerts

See alerts generated for devices across multiple IoT Hubs in [Alerts tab of the IoT Edge fleet view workbook](#).

Click the alert rule name to see more context about the alert. Clicking the device name link will show you the detailed metrics for the device around the time when the alert fired.

Next steps

Enhance your monitoring solution with [metrics from custom modules](#).

Add custom metrics

Article • 12/15/2022

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Gather custom metrics from your IoT Edge modules in addition to the built-in metrics that the system modules provide. The [built-in metrics](#) provide great baseline visibility into your deployment health. However, you may require additional information from custom modules to complete the picture. Custom modules can be integrated into your monitoring solution by using the appropriate [Prometheus client library](#) to emit metrics. This additional information can enable new views or alerts specialized to your requirements.

Sample modules repository

See the [azure-samples repo](#) for examples of custom modules instrumented to emit metrics. Even if a sample in your language of choice isn't yet available, the general approach may help you.

Naming conventions

Consult the [best practices](#) from Prometheus docs for general guidance. The following additional recommendations can be helpful for IoT Edge scenarios.

- Include the module name at the beginning of metric name to make clear which module has emitted the metric.
- Include the IoT hub name or IoT Central application name, IoT Edge device ID, and module ID as labels (also called *tags/dimensions*) in every metric. This information is available as environment variables to every module started by the IoT Edge agent. The approach is [demonstrated](#) by the example in samples repo. Without this context, it's impossible to associate a given metric value to a particular device.

- Include an instance ID in the labels. An instance ID can be any unique ID like a [GUID](#) that is generated during module startup. Instance ID information can help reconcile module restarts when processing a module's metrics in the backend.

Configure the metrics collector to collect custom metrics

Once a custom module is emitting metrics, the next step is to configure the [metrics-collector module](#) to collect and transport custom metrics.

The environment variable `MetricsEndpointsCSV` must be updated to include the URL of the custom module's metrics endpoint. When updating the environment variable, be sure to include the system module endpoints as shown in the [metric collector configuration](#) example.

Note

By default, a custom module's metrics endpoint does not need to be mapped to a host port to allow the metrics collector to access it. Unless explicitly overridden, on Linux, both modules are started on a [user-defined Docker bridge network](#) named *azure-iot-edge*.

User-defined Docker networks include a default DNS resolver that allows inter-module communication using module (container) names. For example, if a custom module named *module1* is emitting metrics on http port 9600 at path */metrics*, the collector should be configured to collect from endpoint `http://module1:9600/metrics`.

Run the following command on the IoT Edge device to test if metrics emitted by a custom module on http port 9600 at path */metrics* are accessible:

Bash

```
sudo docker exec replace-with-metrics-collector-module-name curl http://replace-with-custom-module-name:9600/metrics
```

Add custom visualizations

Once you're receiving custom metrics in Log Analytics, you can create custom visualizations and alerts. The monitoring workbooks can be augmented to add query-

backed visualizations.

Every metric is associated with the resource ID of the IoT hub or IoT Central application. That's why you can check if your custom metrics ingested correctly from the [Logs](#) page of the associated IoT hub or IoT Central application instead of the backing Log Analytics workspace. Use this basic KQL query to verify:

KQL

```
InsightsMetrics  
| where Name == 'replace-with-custom-metric-name'
```

Once you have confirmed ingestion, you can either create a new workbook or augment an existing workbook. Use [workbooks docs](#) and queries from the curated [IoT Edge workbooks](#) as a guide.

When happy with the results, you can [share the workbook](#) with your team or [deploy them programmatically](#) as part of your organization's resource deployments.

Next steps

Explore additional metrics visualization options with [curated workbooks](#).

FAQ and troubleshooting

Article • 02/08/2022

Collector module is unable to collect metrics from built-in endpoints

Check if modules are on the same Docker network

The metrics-collector module relies on Docker's embedded DNS resolver for user-defined networks. The DNS resolver provides the IP address for metrics endpoints that include module name. For example, `http://edgeHub:9600/metrics`.

When modules aren't running in the same network namespace, this mechanism will fail. For instance, some scenarios require running modules on the host network. Collection fails in such scenarios if metrics-collector module is on a different network.

Verify that `httpSettings_enabled` environment variable isn't set to `false`

The built-in metrics endpoints exposed by IoT Edge system modules use http protocol. They won't be available, even within the module network, if http is explicitly disabled via the environment variable setting on Edge Hub or Edge Agent modules.

Set `NO_PROXY` environment variable if using http proxy server

For more information, see [proxy considerations](#).

Update Moby-engine

On Linux hosts, ensure you're using a recent version of the container engine. We recommend updating to the latest version by following the [installation instructions](#).

How do I collect logs along with metrics?

You could use [built-in log pull features](#). A sample solution that uses the built-in log retrieval features is available at <https://aka.ms/iot-elms>.

Why can't I see device metrics in the metrics page in Azure portal?

Azure Monitor's [native metrics](#) technology doesn't yet support Prometheus data format directly. Log-based metrics are currently better suited for IoT Edge metrics because of:

- Native support for Prometheus metrics format via the standard *InsightsMetrics* table.
- Advanced data processing via [KQL](#) for visualizations and alerts.

The use of Log Analytics as the metrics database is the reason why metrics appear in the **Logs** page in Azure portal rather than **Metrics**.

How do I configure metrics-collector in a layered deployment?

The metrics collector doesn't have any service discovery functionality. We recommend including the module in the base or *lower* deployment layer. Include all metrics endpoints that the module might be deployed with in the module's configuration. If a module doesn't appear in a final deployment but its endpoint appears in the collection list, the collector will try to collect, fail, and move on.

Configure an IoT Edge device to act as a transparent gateway

Article • 08/07/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides detailed instructions for configuring an IoT Edge device to function as a transparent gateway for other devices to communicate with IoT Hub. This article uses the term *IoT Edge gateway* to refer to an IoT Edge device configured as a transparent gateway. For more information, see [How an IoT Edge device can be used as a gateway](#).

Note

Downstream devices can't use file upload.

There are three general steps to set up a successful transparent gateway connection. This article covers the first step:

1. Configure the gateway device as a server so that downstream devices can connect to it securely. Set up the gateway to receive messages from downstream devices and route them to the proper destination.
2. Create a device identity for the downstream device so that it can authenticate with IoT Hub. Configure the downstream device to send messages through the gateway device. For those steps, see [Authenticate a downstream device to Azure IoT Hub](#).
3. Connect the downstream device to the gateway device and start sending messages. For those steps, see [Connect a downstream device to an Azure IoT Edge gateway](#).

For a device to act as a gateway, it needs to securely connect to its downstream devices. Azure IoT Edge allows you to use a public key infrastructure (PKI) to set up secure connections between devices. In this case, we're allowing a downstream device to connect to an IoT Edge device acting as a transparent gateway. To maintain reasonable

security, the downstream device should confirm the identity of the gateway device. This identity check prevents your devices from connecting to potentially malicious gateways.

A downstream device can be any application or platform that has an identity created with the [Azure IoT Hub](#) cloud service. These applications often use the [Azure IoT device SDK](#). A downstream device could even be an application running on the IoT Edge gateway device itself.

You can create any certificate infrastructure that enables the trust required for your device-gateway topology. In this article, we assume the same certificate setup that you would use to enable [X.509 CA security](#) in IoT Hub, which involves an X.509 CA certificate associated to a specific IoT hub (the IoT hub root CA), a series of certificates signed with this CA, and a CA for the IoT Edge device.

Note

The term *root CA certificate* used throughout these articles refers to the topmost authority public certificate of the PKI certificate chain, and not necessarily the certificate root of a syndicated certificate authority. In many cases, it is actually an intermediate CA public certificate.

The following steps walk you through the process of creating the certificates and installing them in the right places on the gateway. You can use any machine to generate the certificates, and then copy them over to your IoT Edge device.

Prerequisites

IoT Edge

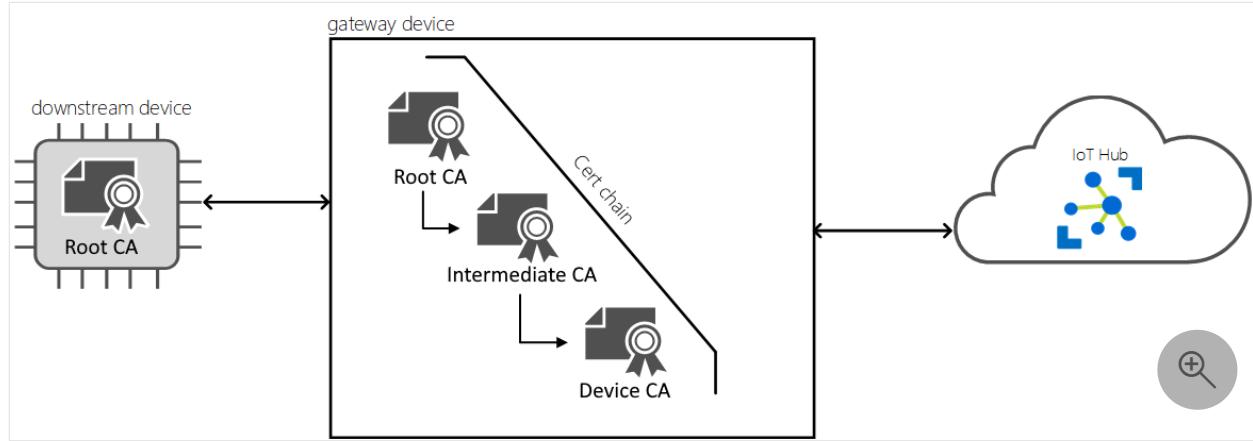
A Linux or Windows device with IoT Edge installed.

If you don't have a device ready, you can create one in an Azure virtual machine. Follow the steps in [Deploy your first IoT Edge module to a virtual Linux device](#) to create an IoT Hub, create a virtual machine, and configure the IoT Edge runtime.

Set up the Edge CA certificate

All IoT Edge gateways need an Edge CA certificate installed on them. The IoT Edge security daemon uses the Edge CA certificate to sign a workload CA certificate, which in turn signs a server certificate for IoT Edge hub. The gateway presents its server

certificate to the downstream device during the initiation of the connection. The downstream device checks to make sure that the server certificate is part of a certificate chain that rolls up to the root CA certificate. This process allows the downstream device to confirm that the gateway comes from a trusted source. For more information, see [Understand how Azure IoT Edge uses certificates](#).



The root CA certificate and the Edge CA certificate (with its private key) need to be present on the IoT Edge gateway device and configured in the IoT Edge config file. Remember that in this case *root CA certificate* means the topmost certificate authority for this IoT Edge scenario. The gateway Edge CA certificate and the downstream device certificates need to roll up to the same root CA certificate.

Tip

The process of installing the root CA certificate and Edge CA certificate on an IoT Edge device is also explained in more detail in [Manage certificates on an IoT Edge device](#).

Have the following files ready:

- Root CA certificate
- Edge CA certificate
- Device CA private key

For production scenarios, you should generate these files with your own certificate authority. For development and test scenarios, you can use demo certificates.

Create demo certificates

If you don't have your own certificate authority and want to use demo certificates, follow the instructions in [Create demo certificates to test IoT Edge device features](#) to create your files. On that page, you need to take the following steps:

1. To start, set up the scripts for generating certificates on your device.
2. Create a root CA certificate. At the end of those instructions, you'll have a root CA certificate file <path>/certs/azure-iot-test-only.root.ca.cert.pem.
3. Create Edge CA certificates. At the end of those instructions, you'll have an Edge CA certificate <path>/certs/iot-edge-device-ca-<cert name>-full-chain.cert.pem and its private key <path>/private/iot-edge-device-ca-<cert name>.key.pem.

Copy certificates to device

IoT Edge

1. Check the certificate meets format requirements.
2. If you created the certificates on a different machine, copy them over to your IoT Edge device. You can use a USB drive, a service like [Azure Key Vault](#), or with a function like [Secure file copy ↗](#).
3. Move the files to the preferred directory for certificates and keys. Use /var/aziot/certs for certificates and /var/aziot/secrets for keys.
4. Create the certificates and keys directories and set permissions. You should store your certificates and keys to the preferred /var/aziot directory. Use /var/aziot/certs for certificates and /var/aziot/secrets for keys.

Bash

```
# If the certificate and keys directories don't exist, create, set ownership, and set permissions
sudo mkdir -p /var/aziot/certs
sudo chown aziotcs:aziotcs /var/aziot/certs
sudo chmod 755 /var/aziot/certs

sudo mkdir -p /var/aziot/secrets
sudo chown aziotks:aziotks /var/aziot/secrets
sudo chmod 700 /var/aziot/secrets
```

5. Change the ownership and permissions of the certificates and keys.

Bash

```
# Give aziotcs ownership to certificates
# Read and write for aziotcs, read-only for others
sudo chown -R aziotcs:aziotcs /var/aziot/certs
sudo find /var/aziot/certs -type f -name "*.*" -exec chmod 644 {}
```

```
\;

# Give aziotks ownership to private keys
# Read and write for aziotks, no permission for others
sudo chown -R aziotks:aziotks /var/aziot/secrets
  sudo find /var/aziot/secrets -type f -name "*.*" -exec chmod 600
{} \;
```

Configure certificates on device

1. On your IoT Edge device, open the config file: `/etc/aziot/config.toml`. If you're using IoT Edge for Linux on Windows, you'll have to connect to the EFLOW virtual machine using the `Connect-EflowVm` PowerShell cmdlet.

💡 Tip

If the config file doesn't exist on your device yet, then use `/etc/aziot/config.toml.edge.template` as a template to create one.

2. Find the `trust_bundle_cert` parameter. Uncomment this line and provide the file URI to the root CA certificate file on your device.
3. Find the `[edge_ca]` section of the file. Uncomment the three lines in this section and provide the file URIs to your certificate and key files as values for the following properties:
 - `cert`: Edge CA certificate
 - `pk`: device CA private key
4. Save and close the file.
5. Apply your changes.

Bash

```
sudo iotedge config apply
```

Deploy edgeHub and route messages

Downstream devices send telemetry and messages to the gateway device, where the IoT Edge hub module is responsible for routing the information to other modules or to IoT

Hub. To prepare your gateway device for this function, make sure that:

- The IoT Edge hub module is deployed to the device.

When you first install IoT Edge on a device, only one system module starts automatically: the IoT Edge agent. Once you create the first deployment for a device, the second system module and the IoT Edge hub start as well. If the **edgeHub** module isn't running on your device, create a deployment for your device.

- The IoT Edge hub module has routes set up to handle incoming messages from downstream devices.

The gateway device must have a route in place to handle messages from downstream devices or else those messages will not be processed. You can send the messages to modules on the gateway device or directly to IoT Hub.

To deploy the IoT Edge hub module and configure it with routes to handle incoming messages from downstream devices, follow these steps:

1. In the Azure portal, navigate to your IoT hub.
2. Go to **Devices** under the **Device management** menu and select your IoT Edge device that you want to use as a gateway.
3. Select **Set Modules**.
4. On the **Modules** page, you can add any modules you want to deploy to the gateway device. For the purposes of this article we're focused on configuring and deploying the **edgeHub** module, which doesn't need to be explicitly set on this page.
5. Select **Next: Routes**.
6. On the **Routes** page, make sure that there is a route to handle messages coming from downstream devices. For example:
 - A route that sends all messages, whether from a module or from a downstream device, to IoT Hub:
 - **Name:** `allMessagesToHub`
 - **Value:** `FROM /messages/* INTO $upstream`
 - A route that sends all messages from all downstream devices to IoT Hub:
 - **Name:** `allDownstreamToHub`

- Value: FROM /messages/* WHERE NOT IS_DEFINED (\$connectionModuleId)
INTO \$upstream

This route works because, unlike messages from IoT Edge modules, messages from downstream devices don't have a module ID associated with them. Using the **WHERE** clause of the route allows us to filter out any messages with that system property.

For more information about message routing, see [Deploy modules and establish routes](#).

7. Once your route or routes are created, select **Review + create**.

8. On the **Review + create** page, select **Create**.

Open ports on gateway device

Standard IoT Edge devices don't need any inbound connectivity to function, because all communication with IoT Hub is done through outbound connections. Gateway devices are different because they need to receive messages from their downstream devices. If a firewall is between the downstream devices and the gateway device, then communication needs to be possible through the firewall as well.

IoT Edge

For a gateway scenario to work, at least one of the IoT Edge Hub's supported protocols must be open for inbound traffic from downstream devices. The supported protocols are MQTT, AMQP, HTTPS, MQTT over WebSockets, and AMQP over WebSockets.

 [Expand table](#)

Port	Protocol
8883	MQTT
5671	AMQP
443	HTTPS MQTT+WS AMQP+WS

Next steps

Now that you have an IoT Edge device set up as a transparent gateway, you need to configure your downstream devices to trust the gateway and send messages to it.

Continue on to [Authenticate a downstream device to Azure IoT Hub](#) for the next steps in setting up your transparent gateway scenario.

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Authenticate a downstream device to Azure IoT Hub

Article • 05/29/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

In a transparent gateway scenario, downstream devices (sometimes called child devices) need identities in IoT Hub like any other device. This article walks through the options for authenticating a downstream device to IoT Hub, and then demonstrates how to declare the gateway connection.

Note

A downstream device emits data directly to the Internet or to gateway devices (IoT Edge-enabled or not). A child device can be a downstream device or a gateway device in a nested topology.

There are three general steps to set up a successful transparent gateway connection. This article covers the second step:

1. Configure the gateway device as a server so that downstream devices can connect to it securely. Set up the gateway to receive messages from downstream devices and route them to the proper destination. For those steps, see [Configure an IoT Edge device to act as a transparent gateway](#).
2. Create a device identity for the downstream device so that it can authenticate with IoT Hub. Configure the downstream device to send messages through the gateway device.
3. Connect the downstream device to the gateway device and start sending messages. For those steps, see [Connect a downstream device to an Azure IoT Edge gateway](#).

Downstream devices can authenticate with IoT Hub using one of three methods: symmetric keys (sometimes referred to as shared access keys), X.509 self-signed

certificates, or X.509 certificate authority (CA) signed certificates. The authentication steps are similar to the steps used to set up any non-IoT-Edge device with IoT Hub, with small differences to declare the gateway relationship.

Automatic provisioning downstream devices with the Azure IoT Hub Device Provisioning Service (DPS) is not supported.

Prerequisites

Complete the steps in [Configure an IoT Edge device to act as a transparent gateway](#).

If you're using X.509 authentication, you will generate certificates for your downstream device. Have the same root CA certificate and the certificate generating script that you used for the transparent gateway article available to use again.

This article refers to the *gateway hostname* at several points. The gateway hostname is declared in the **hostname** parameter of the config file on the IoT Edge gateway device. It's referred to in the connection string of the downstream device. The gateway hostname needs to be resolvable to an IP Address, either using DNS or a host file entry on the downstream device.

Register device with IoT Hub

Choose how you want your downstream device to authenticate with IoT Hub:

- [Symmetric key authentication](#): IoT Hub creates a key that you put on the downstream device. When the device authenticates, IoT Hub checks that the two keys match. You don't need to create additional certificates to use symmetric key authentication.

This method is quicker to get started if you're testing gateways in a development or test scenario.

- [X.509 self-signed authentication](#): Sometimes called thumbprint authentication, because you share the thumbprint from the device's X.509 certificate with IoT Hub.

Certificate authentication is recommended for devices in production scenarios.

- [X.509 CA-signed authentication](#): Upload the root CA certificate to IoT Hub. When devices present their X.509 certificate for authentication, IoT Hub checks that it belongs to a chain of trust signed by the same root CA certificate.

Certificate authentication is recommended for devices in production scenarios.

Symmetric key authentication

Symmetric key authentication, or shared access key authentication, is the simplest way to authenticate with IoT Hub. With symmetric key authentication, a base64 key is associated with your IoT device ID in IoT Hub. You include that key in your IoT applications so that your device can present it when it connects to IoT Hub.

Add a new IoT device in your IoT hub, using either the Azure portal, Azure CLI, or the IoT extension for Visual Studio Code. Remember that downstream devices need to be identified in IoT Hub as regular IoT devices, not IoT Edge devices.

When you create the new device identity, provide the following information:

- Create an ID for your device.
- Select **Symmetric key** as the authentication type.
- Select **Set a parent device** and select the IoT Edge gateway device that this downstream device will connect through. You can always change the parent later.



Create a device

□ X

* Device ID ⓘ

The ID of the new device

Authentication type ⓘ

Symmetric key

X.509 Self-Signed

X.509 CA Signed

* Primary key ⓘ

Enter your primary key

* Secondary key ⓘ

Enter your secondary key

Auto-generate keys ⓘ



Connect this device to an IoT hub ⓘ

Enable

Disable

Parent device ⓘ

No parent device

Set a parent device

Save

ⓘ Note

Setting the parent device used to be an optional step for downstream devices that use symmetric key authentication. However, starting with IoT Edge version 1.1.0 every downstream device must be assigned to a parent device.

You can configure the IoT Edge hub to go back to the previous behavior by setting the environment variable **AuthenticationMode** to the value **CloudAndScope**.

You also can use the [IoT extension for Azure CLI](#) to complete the same operation. The following example uses the `az iot hub device-identity` command to create a new IoT device with symmetric key authentication and assign a parent device:

Azure CLI

```
az iot hub device-identity create -n {iothub name} -d {new device ID} --device-scope {deviceScope of parent device}
```

💡 Tip

You can list device properties including device scope using `az iot hub device-identity list --hub-name {iothub name}`.

Next, [Retrieve and modify the connection string](#) so that your device knows to connect via its gateway.

X.509 self-signed authentication

For X.509 self-signed authentication, sometimes referred to as thumbprint authentication, you need to create certificates to place on your downstream device. These certificates have a thumbprint in them that you share with IoT Hub for authentication.

1. Using your CA certificate, create two device certificates (primary and secondary) for the downstream device.

If you don't have a certificate authority to create X.509 certificates, you can use the IoT Edge demo certificate scripts to [Create downstream device certificates](#). Follow the steps for creating self-signed certificates. Use the same root CA certificate that generated the certificates for your gateway device.

If you create your own certificates, make sure that the device certificate's subject name is set to the device ID that you use when registering the IoT device in the Azure IoT Hub. This setting is required for authentication.

2. Retrieve the SHA1 fingerprint (called a thumbprint in the IoT Hub interface) from each certificate, which is a 40 hexadecimal character string. Use the following openssl command to view the certificate and find the fingerprint:

- Windows:

PowerShell

```
openssl x509 -in <path to primary device certificate>.cert.pem -text -fingerprint
```

- Linux:

Bash

```
openssl x509 -in <path to primary device certificate>.cert.pem -  
text -fingerprint | sed 's/://g'
```

Run this command twice, once for the primary certificate and once for the secondary certificate. You provide fingerprints for both certificates when you register a new IoT device using self-signed X.509 certificates.

3. Navigate to your IoT hub in the Azure portal and create a new IoT device identity with the following values:

- Provide the **Device ID** that matches the subject name of your device certificates.
- Select **X.509 Self-Signed** as the authentication type.
- Paste the hexadecimal strings that you copied from your device's primary and secondary certificates.
- Select **Set a parent device** and choose the IoT Edge gateway device that this downstream device will connect through. You can always change the parent later.



Create a device

□ X

* Device ID ⓘ

The ID of the new device

Authentication type ⓘ

Symmetric key

X.509 Self-Signed

X.509 CA Signed

* Primary Thumbprint ⓘ

Enter your primary thumbprint here

* Secondary Thumbprint ⓘ

Enter your secondary thumbprint here

Connect this device to an IoT hub ⓘ

Enable

Disable

Parent device ⓘ

No parent device

Set a parent device

Save

4. Copy both the primary and secondary device certificates and their keys to any location on the downstream device. Also move a copy of the shared root CA certificate that generated both the gateway device certificate and the downstream device certificates.

You'll reference these certificate files in any applications on the downstream device that connect to IoT Hub. You can use a service like [Azure Key Vault](#) or a function like [Secure copy protocol](#) ↗ to move the certificate files.

5. Depending on your preferred language, review samples of how X.509 certificates can be referenced in IoT applications:

- C#: [Set up X.509 security in your Azure IoT hub](#)
- C: [iotedge_downstream_device_sample.c](#) ↗
- Node.js: [simple_sample_device_x509.js](#) ↗
- Java: [SendEventX509.java](#) ↗

- Python: [send_message_x509.py](#)

You also can use the [IoT extension for Azure CLI](#) to complete the same device creation operation. The following example uses the `az iot hub device-identity` command to create a new IoT device with X.509 self-signed authentication and assigns a parent device:

Azure CLI

```
az iot hub device-identity create -n {iothub name} -d {device ID} --device-scope {deviceScope of gateway device} --am x509_thumbprint --ptp {primary thumbprint} --stp {secondary thumbprint}
```

Tip

You can list device properties including device scope using `az iot hub device-identity list --hub-name {iothub name}`.

Next, [Retrieve and modify the connection string](#) so that your device knows to connect via its gateway.

X.509 CA-signed authentication

For X.509 certificate authority (CA) signed authentication, you need a root CA certificate registered in IoT Hub that you use to sign certificates for your downstream device. Any device using a certificate that was issued by the root CA certificate or any of its intermediate certificates will be permitted to authenticate.

This section is based on the IoT Hub X.509 certificate tutorial series. See [Understanding Public Key Cryptography and X.509 Public Key Infrastructure](#) for the introduction of this series.

1. Using your CA certificate, create two device certificates (primary and secondary) for the downstream device.

If you don't have a certificate authority to create X.509 certificates, you can use the IoT Edge demo certificate scripts to [Create downstream device certificates](#). Follow the steps for creating CA-signed certificates. Use the same root CA certificate that generated the certificates for your gateway device.

2. Follow the instructions in the [Demonstrate proof of possession](#) section of *Set up X.509 security in your Azure IoT hub*. In that section, you perform the following

steps:

- a. Upload a root CA certificate. If you're using the demo certificates, the root CA is <path>/certs/azure-iot-test-only.root.ca.cert.pem.
 - b. Verify that you own that root CA certificate.
3. Follow the instructions in the [Create a device in your IoT Hub](#) section of *Set up X.509 security in your Azure IoT hub*. In that section, you perform the following steps:
- a. Add a new device. Provide a lowercase name for **device ID**, and choose the authentication type **X.509 CA Signed**.
 - b. Set a parent device. Select **Set a parent device** and choose the IoT Edge gateway device that will provide the connection to IoT Hub.
4. Create a certificate chain for your downstream device. Use the same root CA certificate that you uploaded to IoT Hub to make this chain. Use the same lowercase device ID that you gave to your device identity in the portal.
5. Copy the device certificate and keys to any location on the downstream device. Also move a copy of the shared root CA certificate that generated both the gateway device certificate and the downstream device certificates.
- You'll reference these files in any applications on the downstream device that connect to IoT Hub. You can use a service like [Azure Key Vault](#) or a function like [Secure copy protocol](#) to move the certificate files.
6. Depending on your preferred language, review samples of how X.509 certificates can be referenced in IoT applications:

- C#: [Set up X.509 security in your Azure IoT hub](#)
- C: [iotedge_downstream_device_sample.c](#)
- Node.js: [simple_sample_device_x509.js](#)
- Java: [SendEventX509.java](#)
- Python: [send_message_x509.py](#)

You also can use the [IoT extension for Azure CLI](#) to complete the same device creation operation. The following example uses the `az iot hub device-identity` command to create a new IoT device with X.509 CA signed authentication and assigns a parent device:

```
az iot hub device-identity create -n {iothub name} -d {device ID} --device-scope {deviceScope of gateway device} --am x509_ca
```

Tip

You can list device properties including device scope using `az iot hub device-identity list --hub-name {iothub name}`.

Next, [Retrieve and modify the connection string](#) so that your device knows to connect via its gateway.

Retrieve and modify connection string

After creating an IoT device identity in the portal, you can retrieve its primary or secondary keys. One of these keys needs to be included in the connection string that applications use to communicate with IoT Hub. For symmetric key authentication, IoT Hub provides the fully formed connection string in the device details for your convenience. You need to add extra information about the gateway device to the connection string.

Connection strings for downstream devices need the following components:

- The IoT hub that the device connects to: `Hostname={iothub name}.azure-devices.net`
- The device ID registered with the hub: `DeviceID={device ID}`
- The authentication method, whether symmetric key or X.509 certificates
 - If using symmetric key authentication provide either the primary or secondary key: `SharedAccessKey={key}`
 - If using X.509 certificate authentication, provide a flag: `x509=true`
- The gateway device that the device connects through. Provide the **hostname** value from the IoT Edge gateway device's config file: `GatewayHostName={gateway hostname}`

All together, a complete connection string looks like:

Console

```
HostName=myiothub.azure-devices.net;DeviceId=myDownstreamDevice;SharedAccessKey=xxxxyyzzz;GatewayHostName=myGatewayDevice
```

Or:

Console

```
HostName=myiothub.azure-
devices.net;DeviceId=myDownstreamDevice;x509=true;GatewayHostName=myGatewayD
evice
```

Thanks to the parent/child relationship, you can simplify the connection string by calling the gateway directly as the connection host. For example:

Console

```
HostName=myGatewayDevice;DeviceId=myDownstreamDevice;SharedAccessKey=xxxxyyzz
zz
```

You'll use this modified connection string in the next article of the transparent gateway series.

Next steps

At this point, you have an IoT Edge device registered with your IoT hub and configured as a transparent gateway. You also have a downstream device registered with your IoT hub and pointing to its gateway device.

Next, you need to configure your downstream device to trust the gateway device and connect to it securely. Continue on to the next article in the transparent gateway series, [Connect a downstream device to an Azure IoT Edge gateway](#).

Connect a downstream device to an Azure IoT Edge gateway

Article • 08/07/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Here, you find instructions for establishing a trusted connection between downstream devices and IoT Edge transparent [gateways](#). In a transparent gateway scenario, one or more devices can pass their messages through a single gateway device that maintains the connection to IoT Hub. Here, the terms *gateway* and *IoT Edge gateway* refer to an IoT Edge device configured as a transparent gateway.

Note

A downstream device emits data directly to the Internet or to gateway devices (IoT Edge-enabled or not). A child device can be a downstream device or a gateway device in a nested topology.

There are three general steps to set up a successful transparent gateway connection. This article explains the third step.

1. Configure the gateway device as a server so that downstream devices can connect to it securely. Set up the gateway to receive messages from downstream devices and route them to the proper destination. For those steps, see [Configure an IoT Edge device to act as a transparent gateway](#).
2. Create a device identity for the downstream device so that it can authenticate with IoT Hub. Configure the downstream device to send messages through the gateway device. For those steps, see [Authenticate a downstream device to Azure IoT Hub](#).
3. **Connect the downstream device to the gateway device and start sending messages.**

This article helps you understand downstream device connection components, such as:

- Transport layer security (TLS) and certificate fundamentals.
- TLS libraries working across different operating systems that handle certificates differently.

You then walk through Azure IoT samples, in your preferred language, to get your device to send messages to the gateway.

Prerequisites

Acquire the following to prepare your downstream device:

- A downstream device.

This device can be any application or platform that has an identity created with the Azure IoT Hub cloud service. In many cases, these applications use the [Azure IoT device SDK](#). A downstream device can also be an application running on the IoT Edge gateway device itself.

Later, this article provides the steps for connecting an *IoT* device as a downstream device. If you prefer to use an *IoT Edge* device as a downstream device, see [Connect Azure IoT Edge devices together to create a hierarchy \(nested edge\)](#).

- A root CA certificate file.

This file was used to generate the Edge CA certificate in [Configure an IoT Edge device to act as a transparent gateway](#), which is available on your downstream device.

Your downstream device uses this certificate to validate the identity of the gateway device. This trusted certificate validates the transport layer security (TLS) connections to the gateway device. See usage details in the [Provide the root CA certificate](#) section.

- A modified connection string that points to the gateway device.

How to modify your connection string is explained in [Authenticate a downstream device to Azure IoT Hub](#).

Note

IoT devices registered with IoT Hub can use [module twins](#) to isolate different processes, hardware, or functions on a single device. IoT Edge gateways support

downstream module connections, using symmetric key authentication but not X.509 certificate authentication.

Understand TLS and certificate fundamentals

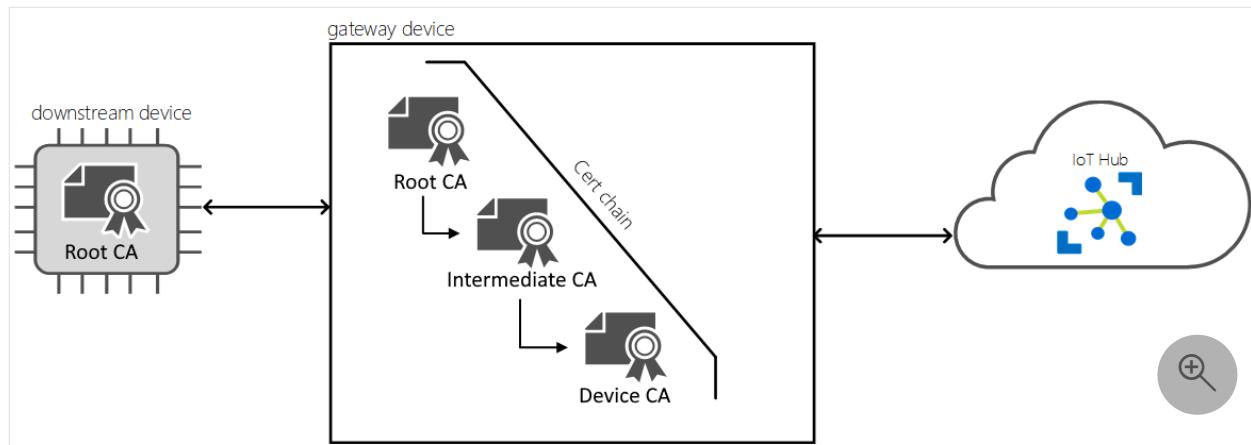
The challenge of securely connecting downstream devices to IoT Edge is just like any other secure client/server communication that occurs over the internet. A client and a server securely communicate over the internet using [Transport layer security \(TLS\)](#). TLS is built using standard [Public key infrastructure \(PKI\)](#) constructs called certificates. TLS is a fairly involved specification and addresses a wide range of topics related to securing two endpoints. This section summarizes the concepts relevant for you to securely connect devices to an IoT Edge gateway.

When a client connects to a server, the server presents a chain of certificates, called the *server certificate chain*. A certificate chain typically comprises a root certificate authority (CA) certificate, one or more intermediate CA certificates, and finally the server's certificate itself. A client establishes trust with a server by cryptographically verifying the entire server certificate chain. This client validation of the server certificate chain is called *server chain validation*. The client challenges the server to prove possession of the private key associated with the server certificate in a process called *proof of possession*. The combination of server chain validation and proof of possession is called *server authentication*. To validate a server certificate chain, a client needs a copy of the root CA certificate that was used to create (or issue) the server's certificate. Normally when connecting to websites, a browser comes preconfigured with commonly used CA certificates so the client has a seamless process.

When a device connects to Azure IoT Hub, the device is the client and the IoT Hub cloud service is the server. The IoT Hub cloud service is backed by a root CA certificate called **Baltimore CyberTrust Root**, which is publicly available and widely used. Since the IoT Hub CA certificate is already installed on most devices, many TLS implementations (OpenSSL, Schannel, LibreSSL) automatically use it during server certificate validation. However, a device that successfully connects to IoT Hub may have issues trying to connect to an IoT Edge gateway.

When a device connects to an IoT Edge gateway, the downstream device is the client and the gateway device is the server. Azure IoT Edge allows you to build gateway certificate chains however they see fit. You may choose to use a public CA certificate, like Baltimore, or use a self-signed (or in-house) root CA certificate. Public CA certificates often have a cost associated with them, so are typically used in production scenarios. Self-signed CA certificates are preferred for development and testing. The demo certificates are self-signed root CA certificates.

When you use a self-signed root CA certificate for an IoT Edge gateway, it needs to be installed on or provided to all the downstream devices attempting to connect to the gateway.



To learn more about IoT Edge certificates and some production implications, see [IoT Edge certificate usage details](#).

Provide the root CA certificate

To verify the gateway device's certificates, the downstream device needs its own copy of the root CA certificate. If you used the scripts provided in the IoT Edge git repository to create test certificates, then the root CA certificate is called `azure-iot-test-only.root.ca.cert.pem`.

If you haven't already, move this certificate file to any directory on your downstream device. You can move the file by either installing the CA certificate in the operating system's certificate store or (for certain languages) by referencing the certificate within applications using the Azure IoT SDKs.

You can use a service like [Azure Key Vault](#) or a function like [Secure copy protocol](#) to move the certificate file.

Install certificates in the OS

Once the root CA certificate is on the downstream device, make sure the applications that are connecting to the gateway can access the certificate.

Installing the root CA certificate in the operating system's certificate store generally allows most applications to use the root CA certificate. There are some exceptions, like NodeJS applications that don't use the OS certificate store but rather use the Node runtime's internal certificate store. If you can't install the certificate at the operating system level, skip ahead to [Use certificates with Azure IoT SDKs](#).

Install the root CA certificate on either Ubuntu or Windows.

Windows

The following steps are an example of how to install a CA certificate on a Windows host. This example assumes that you're using the `azure-iot-test-only.root.ca.cert.pem` certificate from the prerequisites articles, and that you've copied the certificate into a location on the downstream device.

You can install certificates using PowerShell's [Import-Certificate](#) as an administrator:

PowerShell

```
import-certificate <file path>\azure-iot-test-only.root.ca.cert.pem -  
certstorelocation cert:\LocalMachine\root
```

You can also install certificates using the `certlm` utility:

1. In the Start menu, search for and select **Manage computer certificates**. A utility called `certlm` opens.
2. Navigate to **Certificates - Local Computer > Trusted Root Certification Authorities**.
3. Right-click **Certificates** and select **All Tasks > Import**. The certificate import wizard should launch.
4. Follow the steps as directed and import certificate file `<file path>/azure-iot-test-only.root.ca.cert.pem`. When completed, you should see a "Successfully imported" message.

You can also install certificates programmatically using .NET APIs, as shown in the .NET sample later in this article.

Typically applications use the Windows provided TLS stack called [Schannel](#) to securely connect over TLS. Schannel *requires* certificates to be installed in the Windows certificate store before attempting to establish a TLS connection.

Use certificates with Azure IoT SDKs

[Azure IoT SDKs](#) connect to an IoT Edge device using simple sample applications. The samples' goal is to connect the device client and send telemetry messages to the gateway, then close the connection and exit.

Before using the application-level samples, obtain the following items:

- Your IoT Hub connection string, from your downstream device, modified to point to the gateway device.
- Any certificates required to authenticate your downstream device to IoT Hub. For more information, see [Authenticate a downstream device to Azure IoT Hub](#).
- The full path to the root CA certificate that you copied and saved somewhere on your downstream device.

For example: <file path>/azure-iot-test-only.root.ca.cert.pem.

Now you're ready to use certificates with a sample in the language of your choice:

NodeJS

This section provides a sample application to connect an Azure IoT NodeJS device client to an IoT Edge gateway. For NodeJS applications, you must install the root CA certificate at the application level as shown here. NodeJS applications don't use the system's certificate store.

1. Get the sample for `edge_downstream_device.js` from the [Azure IoT device SDK for Node.js samples repo ↗](#).
2. Make sure that you have all the prerequisites to run the sample by reviewing the `readme.md` file.
3. In the `edge_downstream_device.js` file, update the `connectionString` and `edge_ca_cert_path` variables.
4. Refer to the SDK documentation for instructions on how to run the sample on your device.

To understand the sample that you're running, the following code snippet is how the client SDK reads the certificate file and uses it to establish a secure TLS connection:

JavaScript

```
// Provide the Azure IoT device client via setOptions with the X509
// Edge root CA certificate that was used to setup the Edge runtime
var options = {
  ca : fs.readFileSync(edge_ca_cert_path, 'utf-8'),
};
```

Test the gateway connection

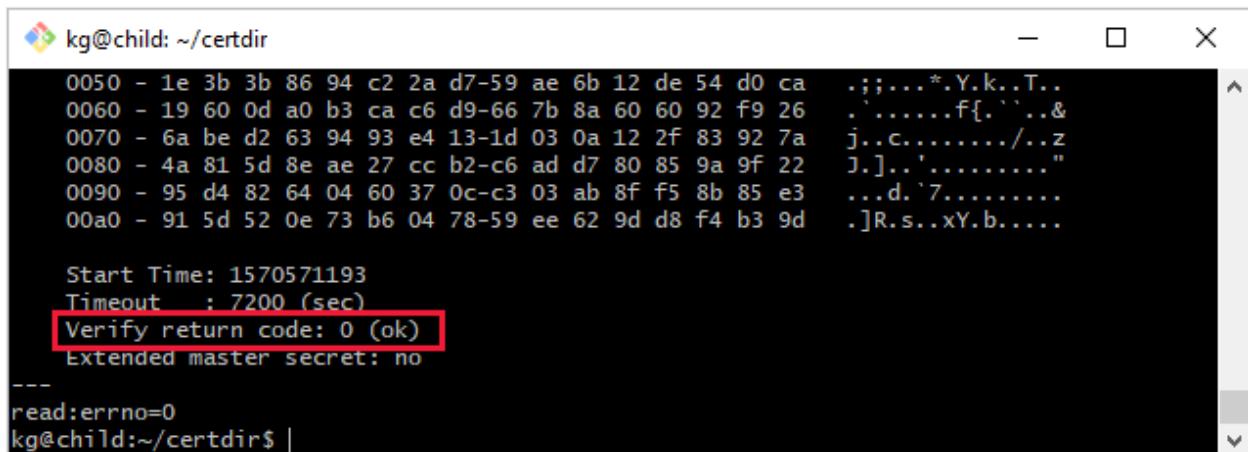
Use this sample command on the downstream device to test that it can connect to the gateway device:

```
cmd/sh  
  
openssl s_client -connect mygateway.contoso.com:8883 -CAfile  
<CERTDIR>/certs/azure-iot-test-only.root.ca.cert.pem -showcerts
```

This command tests connection over MQTTS (port 8883). If you're using a different protocol, adjust the command as necessary for AMQPS (5671) or HTTPS (443).

The output of this command may be long, including information about all the certificates in the chain. If your connection is successful, you see a line like

Verification: OK or Verify return code: 0 (ok).



```
kg@child: ~/certdir  
0050 - 1e 3b 3b 86 94 c2 2a d7-59 ae 6b 12 de 54 d0 ca .;...*.Y.k..T..  
0060 - 19 60 0d a0 b3 ca c6 d9-66 7b 8a 60 60 92 f9 26 .'. ....f{.``..&  
0070 - 6a be d2 63 94 93 e4 13-1d 03 0a 12 2f 83 92 7a j..c...../.z  
0080 - 4a 81 5d 8e ae 27 cc b2-c6 ad d7 80 85 9a 9f 22 J.]..'. ...."  
0090 - 95 d4 82 64 04 60 37 0c-c3 03 ab 8f f5 8b 85 e3 ...d.'7.....  
00a0 - 91 5d 52 0e 73 b6 04 78-59 ee 62 9d d8 f4 b3 9d .]R.s..xY.b.....  
  
Start Time: 1570571193  
Timeout : 7200 (sec)  
Verify return code: 0 (ok) Verify return code: 0 (ok)  
Extended master secret: no  
--  
read:errno=0  
kg@child:~/certdir$ |
```

Troubleshoot the gateway connection

If your downstream device connection to its gateway device is unstable, consider these questions for a resolution.

- Is the gateway hostname in the connection string the same as the hostname value in the IoT Edge config file on the gateway device?
- Is the gateway hostname resolvable to an IP Address? You can resolve intermittent connections either by using DNS or by adding a host file entry on the downstream device.
- Are communication ports open in your firewall? Communication based on the protocol used (MQTTS:8883/AMQPS:5671/HTTPS:433) must be possible between downstream device and the transparent IoT Edge.

Next steps

Learn how IoT Edge can extend [offline capabilities](#) to downstream devices.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Connect Azure IoT Edge devices to create a hierarchy

Article • 08/07/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides steps for establishing a trusted connection between an IoT Edge gateway and a downstream IoT Edge device. This configuration is also known as *nested edge*.

In a gateway scenario, an IoT Edge device can be both a gateway and a downstream device. Multiple IoT Edge gateways can be layered to create a hierarchy of devices. The downstream (child) devices can authenticate and send or receive messages through their gateway (parent) device.

There are two different configurations for IoT Edge devices in a gateway hierarchy, and this article address both. The first is the **top layer** IoT Edge device. When multiple IoT Edge devices are connecting through each other, any device that doesn't have a parent device but connects directly to IoT Hub is considered to be in the top layer. This device is responsible for handling requests from all the devices below it. The other configuration applies to any IoT Edge device in a **lower layer** of the hierarchy. These devices might be a gateway for other downstream IoT and IoT Edge devices, but also need to route any communications through their own parent devices.

Some network architectures require that only the top IoT Edge device in a hierarchy can connect to the cloud. In this configuration, all IoT Edge devices in lower layers of a hierarchy can only communicate with their gateway (parent) device and any downstream (child) devices.

All the steps in this article build on [Configure an IoT Edge device to act as a transparent gateway](#), which sets up an IoT Edge device to be a gateway for downstream IoT devices. The same basic steps apply to all gateway scenarios:

- **Authentication:** Create IoT Hub identities for all devices in the gateway hierarchy.

- **Authorization:** Set up the parent/child relationship in IoT Hub to authorize downstream devices to connect to their parent device like they would connect to IoT Hub.
- **Gateway discovery:** Ensure that the downstream device can find its parent device on the local network.
- **Secure connection:** Establish a secure connection with trusted certificates that are part of the same chain.

Prerequisites

- A free or standard IoT hub.
- At least two **IoT Edge devices**, one to be the top layer device and one or more lower layer devices. If you don't have IoT Edge devices available, you can [Run Azure IoT Edge on Ubuntu virtual machines](#).
- If you use the [Azure CLI](#) to create and manage devices, install the [Azure IoT extension](#).

Tip

This article provides detailed steps and options to help you create the right gateway hierarchy for your scenario. For a guided tutorial, see [Create a hierarchy of IoT Edge devices using gateways](#).

Create a gateway hierarchy

You create an IoT Edge gateway hierarchy by defining parent/child relationships for the IoT Edge devices in the scenario. You can set a parent device when you create a new device identity, or you can manage the parent and children of an existing device identity.

The step of setting up parent/child relationships authorizes downstream devices to connect to their parent device like they would connect to IoT Hub.

Only IoT Edge devices can be parent devices, but both IoT Edge devices and IoT devices can be children. A parent can have many children, but a child can only have one parent. A gateway hierarchy is created by chaining parent/child sets together so that the child of one device is the parent of another.

By default, a parent can have up to 100 children. You can change this limit by setting the **MaxConnectedClients** environment variable in the parent device's edgeHub module.

In the Azure portal, you can manage the parent/child relationship when you create new device identities, or by editing existing devices.

When you create a new IoT Edge device, you have the option of choosing parent and children devices from the list of existing IoT Edge devices in that hub.

1. In the [Azure portal](#), navigate to your IoT hub.
2. Select **Devices** under the **Device management** menu.
3. Select **Add device** then check the **IoT Edge Device** checkbox.
4. Along with setting the device ID and authentication settings, you can **Set a parent device** or **Choose child devices**.
5. Choose the device or devices that you want as a parent or child.

You can also create or manage parent/child relationships for existing devices.

1. In the [Azure portal](#), navigate to your IoT hub.
2. Select **Devices** in the **Device management** menu.
3. Select the **IoT Edge** device you want to manage from the list.
4. Select the **Set a parent device** *gear* icon or **Manage child devices**.
5. Add or remove any parent or child devices.

ⓘ Note

If you wish to establish parent-child relationships programmatically, you can use the C#, Java, or Node.js [IoT Hub Service SDK](#).

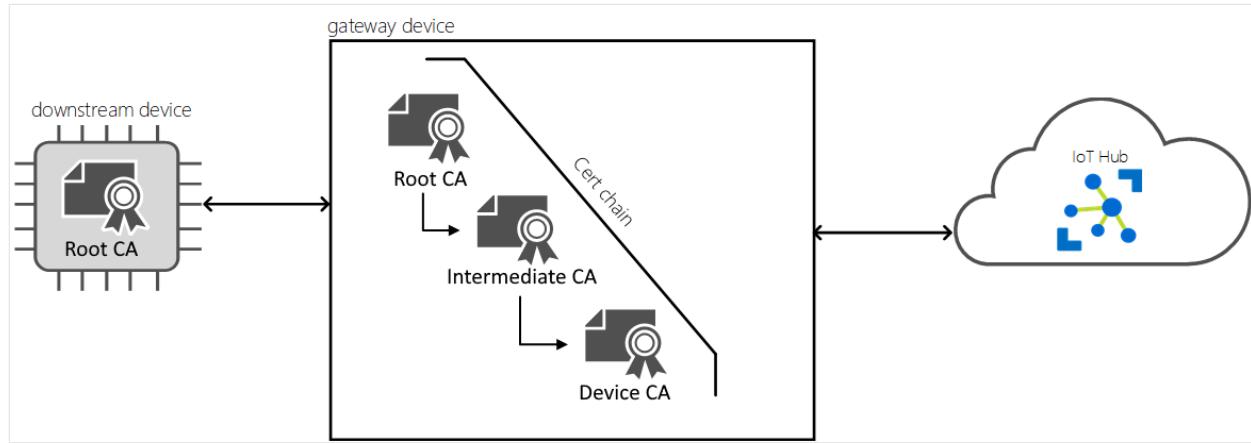
Here is an [example of assigning child devices](#) using the C# SDK. The task `RegistryManager_AddAndRemoveDeviceWithScope()` shows how to programmatically create a three-layer hierarchy. An IoT Edge device is in layer one, as the parent. Another IoT Edge device is in layer two, serving as both a child and a parent. Finally, an IoT device is in layer three, as the lowest layer child device.

Generate certificates

A consistent chain of certificates must be installed across devices in the same gateway hierarchy to establish a secure communication between themselves. Every device in the hierarchy, whether an IoT Edge device or an IoT downstream device, needs a copy of the

same root CA certificate. Each IoT Edge device in the hierarchy then uses that root CA certificate as the root for its Edge CA certificate.

With this setup, each downstream IoT Edge device can verify the identity of their parent by verifying that the *edgeHub* they connect to has a server certificate that is signed by the shared root CA certificate.



For more information about IoT Edge certificate requirements, see [Understand how Azure IoT Edge uses certificates](#).

1. Create or request the following certificates:

- A **root CA certificate**, which is the topmost shared certificate for all the devices in a given gateway hierarchy. This certificate is installed on all devices.
- Any **intermediate certificates** that you want to include in the root certificate chain.
- An **Edge CA certificate** and its **private key**, generated by the root and intermediate certificates. You need one unique Edge CA certificate for each IoT Edge device in the gateway hierarchy.

You can use either a self-signed certificate authority or purchase one from a trusted commercial certificate authority like Baltimore, Verisign, DigiCert, or GlobalSign.

2. If you don't have your own certificates to use for test, create one set of root and intermediate certificates, then create Edge CA certificates for each device. In this article, we'll use test certificates generated using [test CA certificates for samples and tutorials](#). For example, the following commands create a root CA certificate, a parent device certificate, and a child device certificate.

Bash

```
# !!! For test only - do not use in production !!!  
# Create the the root CA test certificate
```

```
./certGen.sh create_root_and_intermediate

# Create the parent (gateway) device test certificate
# signed by the shared root CA certificate
./certGen.sh create_edge_device_ca_certificate "gateway"

# Create the downstream device test certificate
# signed by the shared root CA certificate
./certGen.sh create_edge_device_ca_certificate "downstream"
```

⚠ Warning

Don't use certificates created by the test scripts for production. They contain hard-coded passwords and expire by default after 30 days. The test CA certificates are provided for demonstration purposes to help you understand CA Certificates. Use your own security best practices for certification creation and lifetime management in production.

For more information about creating test certificates, see [create demo certificates to test IoT Edge device features](#).

3. You'll need to transfer the certificates and keys to each device. You can use a USB drive, a service like [Azure Key Vault](#), or with a function like [Secure file copy](#). Choose one of these methods that best matches your scenario. Copy the files to the preferred directory for certificates and keys. Use `/var/aziot/certs` for certificates and `/var/aziot/secrets` for keys.

For more information on installing certificates on a device, see [Manage certificates on an IoT Edge device](#).

Configure parent device

To configure your parent device, open a local or remote command shell.

To enable secure connections, every IoT Edge parent device in a gateway scenario needs to be configured with a unique Edge CA certificate and a copy of the root CA certificate shared by all devices in the gateway hierarchy.

1. Check your certificates meet the [format requirements](#).
2. Transfer the **root CA certificate**, **parent Edge CA certificate**, and **parent private key** to the parent device.

3. Copy the certificates and keys to the correct directories. The preferred directories for device certificates are `/var/aziot/certs` for the certificates and `/var/aziot/secrets` for keys.

Bash

```
### Copy device certificate ###

# If the device certificate and keys directories don't exist, create,
# set ownership, and set permissions
sudo mkdir -p /var/aziot/certs
sudo chown aziotcs:aziotcs /var/aziot/certs
sudo chmod 755 /var/aziot/certs

sudo mkdir -p /var/aziot/secrets
sudo chown aziotks:aziotks /var/aziot/secrets
sudo chmod 700 /var/aziot/secrets

# Copy full-chain device certificate and private key into the correct
# directory
sudo cp iot-edge-device-ca-gateway-full-chain.cert.pem /var/aziot/certs
sudo cp iot-edge-device-ca-gateway.key.pem /var/aziot/secrets

### Root certificate ###

# Copy root certificate into the /certs directory
sudo cp azure-iot-test-only.root.ca.cert.pem /var/aziot/certs

# Copy root certificate into the CA certificate directory and add .crt
# extension.
# The root certificate must be in the CA certificate directory to
# install it in the certificate store.
# Use the appropriate copy command for your device OS or if using
# EFLOW.

# For Ubuntu and Debian, use /usr/local/share/ca-certificates/
sudo cp azure-iot-test-only.root.ca.cert.pem /usr/local/share/azure-
iot-test-only.root.ca.cert.pem.crt
# For EFLOW, use /etc/pki/ca-trust/source/anchors/
sudo cp azure-iot-test-only.root.ca.cert.pem /etc/pki/ca-
trust/source/anchors/azure-iot-test-only.root.ca.pem.crt
```

4. Change the ownership and permissions of the certificates and keys.

Bash

```
# Give aziotcs ownership to certificates
# Read and write for aziotcs, read-only for others
sudo chown -R aziotcs:aziotcs /var/aziot/certs
sudo find /var/aziot/certs -type f -name "*.*" -exec chmod 644 {} \;

# Give aziotks ownership to private keys
```

```

# Read and write for aziotks, no permission for others
sudo chown -R aziotks:aziotks /var/aziot/secrets
sudo find /var/aziot/secrets -type f -name "*.*" -exec chmod 600 {} \;

# Verify permissions of directories and files
sudo ls -Rla /var/aziot

```

The output of list with correct ownership and permission is similar to the following:

Output

```

azureUser@vm:/var/aziot$ sudo ls -Rla /var/aziot
/var/aziot:
total 16
drwxr-xr-x  4 root      root      4096 Dec 14  00:16 .
drwxr-xr-x 15 root      root      4096 Dec 14  00:15 ..
drwxr-xr-x  2 aziotcs  aziotcs  4096 Jan 14  00:31 certs
drwx-----  2 aziotks  aziotks  4096 Jan 23 17:23 secrets

/var/aziot/certs:
total 20
drwxr-xr-x  2 aziotcs  aziotcs  4096 Jan 14  00:31 .
drwxr-xr-x  4 root      root      4096 Dec 14  00:16 ..
-rw-r--r--  1 aziotcs  aziotcs 1984 Jan 14  00:24 azure-iot-test-
only.root.ca.cert.pem
-rw-r--r--  1 aziotcs  aziotcs 5887 Jan 14  00:27 iot-edge-device-ca-
gateway-full-chain.cert.pem

/var/aziot/secrets:
total 16
drwx-----  2 aziotks  aziotks  4096 Jan 23 17:23 .
drwxr-xr-x  4 root      root      4096 Dec 14  00:16 ..
-rw-----  1 aziotks  aziotks 3243 Jan 14  00:28 iot-edge-device-ca-
gateway.key.pem

```

5. Install the **root CA certificate** on the parent IoT Edge device by updating the certificate store on the device using the platform-specific command.

Bash

```

# Update the certificate store

# For Ubuntu or Debian - use update-ca-certificates
sudo update-ca-certificates
# For EFLOW or RHEL - use update-ca-trust
sudo update-ca-trust

```

For more information about using `update-ca-trust` in EFLOW, see [CBL-Mariner SSL CA certificates management](#).

The command reports one certificate was added to `/etc/ssl/certs`.

Output

```
Updating certificates in /etc/ssl/certs...
1 added, 0 removed; done.
```

Update parent configuration file

You should already have IoT Edge installed on your device. If not, follow the steps to [Manually provision a single Linux IoT Edge device](#).

1. Verify the `/etc/aziot/config.toml` configuration file exists on the parent device.

If the config file doesn't exist on your device, use the following command to create it based on the template file:

Bash

```
sudo cp /etc/aziot/config.toml.edge.template /etc/aziot/config.toml
```

You can also use the template file as a reference to add configuration parameters in this section.

2. Open the IoT Edge configuration file using an editor. For example, use the `nano` editor to open the `/etc/aziot/config.toml` file.

Bash

```
sudo nano /etc/aziot/config.toml
```

3. Find the **hostname** parameter or add it to the beginning of the configuration file. Update the value to be the fully qualified domain name (FQDN) or the IP address of the IoT Edge parent device. For example:

toml

```
hostname = "10.0.0.4"
```

To enable gateway discovery, every IoT Edge gateway (parent) device needs to specify a **hostname** parameter that its child devices will use to find it on the local network. Every downstream IoT Edge device needs to specify a **parent_hostname**

parameter to identify its parent. In a hierarchical scenario where a single IoT Edge device is both a parent and a child device, it needs both parameters.

The *hostname* and *trust_bundle_cert* parameters must be at the beginning of the configuration file before any sections. Adding the parameter before defined sections, ensures it's applied correctly.

Use a hostname shorter than 64 characters, which is the character limit for a server certificate common name.

Be consistent with the hostname pattern across a gateway hierarchy. Use either FQDNs or IP addresses, but not both. FQDN or IP address is required to connect downstream devices.

Set the hostname before the *edgeHub* container is created. If *edgeHub* is running, changing the hostname in the configuration file won't take effect until the container is recreated. For more information on how to verify the hostname is applied, see the [verify parent configuration](#) section.

4. Find the **Trust bundle cert** parameter or add it to the beginning of the configuration file.

Update the `trust_bundle_cert` parameter with the file URI to the root CA certificate on your device. For example:

```
toml  
  
trust_bundle_cert = "file:///var/aziot/certs/azure-iot-test-  
only.root.ca.cert.pem"
```

5. Find or add the **Edge CA certificate** section in the config file. Update the certificate `cert` and private key `pk` parameters with the file URI paths for the full-chain certificate and key files on the parent IoT Edge device. IoT Edge requires the certificate and private key to be in text-based privacy-enhanced mail (PEM) format. For example:

```
toml  
  
[edge_ca]  
cert = "file:///var/aziot/certs/iot-edge-device-ca-gateway-full-  
chain.cert.pem"  
pk = "file:///var/aziot/secrets/iot-edge-device-ca-gateway.key.pem"
```

6. Verify your IoT Edge device uses the correct version of the IoT Edge agent when it starts. Find the **Default Edge Agent** section and set the image value for IoT Edge to version 1.5. For example:

```
toml

[agent]
name = "edgeAgent"
type = "docker"

[agent.config]
image = "mcr.microsoft.com/azureiotedge-agent:1.5"
```

7. The beginning of your parent configuration file should look similar to the following example.

```
toml

hostname = "10.0.0.4"
trust_bundle_cert = "file:///var/aziot/certs/azure-iot-test-
only.root.ca.cert.pem"

[edge_ca]
cert = "file:///var/aziot/certs/iot-edge-device-ca-gateway-full-
chain.cert.pem"
pk = "file:///var/aziot/secrets/iot-edge-device-ca-gateway.key.pem"
```

8. Save and close the `config.toml` configuration file. For example if you're using the **nano** editor, select **Ctrl+O - Write Out**, **Enter**, and **Ctrl+X - Exit**.

9. If you've used any other certificates for IoT Edge before, delete the files in the following two directories to make sure that your new certificates get applied:

- `/var/lib/aziot/certd/certs`
- `/var/lib/aziot/keyd/keys`

10. Apply your changes.

```
Bash

sudo iotedge config apply
```

11. Check for any errors in the configuration.

```
Bash
```

```
sudo iotedge check --verbose
```

(!) Note

On a newly provisioned device, you may see an error related to IoT Edge Hub:

- ✗ **production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error**

Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module isn't running. To resolve the error, in IoT Hub, set the modules for the device and create a deployment. Creating a deployment for the device starts the modules on the device including the IoT Edge Hub module.

Verify parent configuration

The *hostname* must be a qualified domain name (FQDN) or the IP address of the IoT Edge device because IoT Edge uses this value in the server certificate when downstream devices connect. The values must match or you'll get *IP address mismatch* error.

To verify the *hostname*, you need to inspect the environment variables of the *edgeHub* container.

1. List the running IoT Edge containers.

Bash

```
iotedge list
```

Verify *edgeAgent* and *edgeHub* containers are running. The command output should be similar to the following example.

Output

NAME	STATUS	DESCRIPTION	CONFIG
SimulatedTemperatureSensor	running	Up 5 seconds	
mcr.microsoft.com.azureiotedge-simulated-temperature-sensor:1.0			
edgeAgent	running	Up 17 seconds	
mcr.microsoft.com.azureiotedge-agent:1.5			
edgeHub	running	Up 6 seconds	
mcr.microsoft.com.azureiotedge-hub:1.5			

2. Inspect the *edgeHub* container.

```
Bash
```

```
sudo docker inspect edgeHub
```

3. In the output, find the **EdgeDeviceHostName** parameter in the *Env* section.

```
JSON
```

```
"EdgeDeviceHostName=10.0.0.4"
```

4. Verify the *EdgeDeviceHostName* parameter value matches the `config.toml` *hostname* setting. If it doesn't match, the *edgeHub* container was running when you modified and applied the configuration. To update the *EdgeDeviceHostName*, remove the *edgeAgent* container.

```
Bash
```

```
sudo docker rm -f edgeAgent
```

The *edgeAgent* and *edgeHub* containers are recreated and started within a few minutes. Once *edgeHub* container is running, inspect the container and verify the *EdgeDeviceHostName* parameter matches the configuration file.

Configure downstream device

To configure your downstream device, open a local or remote command shell.

To enable secure connections, every IoT Edge downstream device in a gateway scenario needs to be configured with a unique Edge CA certificate and a copy of the root CA certificate shared by all devices in the gateway hierarchy.

1. Check your certificates meet the [format requirements](#).
2. Transfer the **root CA certificate**, **child Edge CA certificate**, and **child private key** to the downstream device.
3. Copy the certificates and keys to the correct directories. The preferred directories for device certificates are `/var/aziot/certs` for the certificates and `/var/aziot/secrets` for keys.

```
Bash
```

```

### Copy device certificate ###

# If the device certificate and keys directories don't exist, create,
# set ownership, and set permissions
sudo mkdir -p /var/aziot/certs
sudo chown aziotcs:aziotcs /var/aziot/certs
sudo chmod 755 /var/aziot/certs

sudo mkdir -p /var/aziot/secrets
sudo chown aziotks:aziotks /var/aziot/secrets
sudo chmod 700 /var/aziot/secrets

# Copy device full-chain certificate and private key into the correct
# directory
sudo cp iot-device-downstream-full-chain.cert.pem /var/aziot/certs
sudo cp iot-device-downstream.key.pem /var/aziot/secrets

### Root certificate ###

# Copy root certificate into the /certs directory
sudo cp azure-iot-test-only.root.ca.cert.pem /var/aziot/certs

# Copy root certificate into the CA certificate directory and add .crt
# extension.
# The root certificate must be in the CA certificate directory to
# install it in the certificate store.
# Use the appropriate copy command for your device OS or if using
# EFLOW.

# For Ubuntu and Debian, use /usr/local/share/ca-certificates/
sudo cp azure-iot-test-only.root.ca.cert.pem /usr/local/share/azure-
iot-test-only.root.ca.cert.pem.crt
# For EFLOW, use /etc/pki/ca-trust/source/anchors/
sudo cp azure-iot-test-only.root.ca.cert.pem /etc/pki/ca-
trust/source/anchors/azure-iot-test-only.root.ca.pem.crt

```

4. Change the ownership and permissions of the certificates and keys.

Bash

```

# Give aziotcs ownership to certificates
# Read and write for aziotcs, read-only for others
sudo chown -R aziotcs:aziotcs /var/aziot/certs
sudo find /var/aziot/certs -type f -name "*.*" -exec chmod 644 {} \;

# Give aziotks ownership to private keys
# Read and write for aziotks, no permission for others
sudo chown -R aziotks:aziotks /var/aziot/secrets
sudo find /var/aziot/secrets -type f -name "*.*" -exec chmod 600 {} \;

```

5. Install the **root CA certificate** on the downstream IoT Edge device by updating the certificate store on the device using the platform-specific command.

Bash

```
# Update the certificate store

# For Ubuntu or Debian - use update-ca-certificates
sudo update-ca-certificates
# For EFLOW or RHEL - use update-ca-trust
sudo update-ca-trust
```

For more information about using `update-ca-trust` in EFLOW, see [CBL-Mariner SSL CA certificates management](#).

The command reports one certificate was added to `/etc/ssl/certs`.

Output

```
Updating certificates in /etc/ssl/certs...
1 added, 0 removed; done.
```

Update downstream configuration file

You should already have IoT Edge installed on your device. If not, follow the steps to [Manually provision a single Linux IoT Edge device](#).

1. Verify the `/etc/aziot/config.toml` configuration file exists on the downstream device.

If the config file doesn't exist on your device, use the following command to create it based on the template file:

Bash

```
sudo cp /etc/aziot/config.toml.edge.template /etc/aziot/config.toml
```

You can also use the template file as a reference to add configuration parameters in this section.

2. Open the IoT Edge configuration file using an editor. For example, use the `nano` editor to open the `/etc/aziot/config.toml` file.

Bash

```
sudo nano /etc/aziot/config.toml
```

3. Find the **parent_hostname** parameter or add it to the beginning of the configuration file Every downstream IoT Edge device needs to specify a **parent_hostname** parameter to identify its parent. Update the **parent_hostname** parameter to be the FQDN or IP address of the parent device, matching whatever was provided as the hostname in the parent device's config file. For example:

```
toml
```

```
parent_hostname = "10.0.0.4"
```

4. Find the **Trust bundle cert** parameter or add it to the beginning of the configuration file.

Update the **trust_bundle_cert** parameter with the file URI to the root CA certificate on your device. For example:

```
toml
```

```
trust_bundle_cert = "file:///var/aziot/certs/azure-iot-test-only.root.ca.cert.pem"
```

5. Find or add the **Edge CA certificate** section in the configuration file. Update the certificate **cert** and private key **pk** parameters with the file URI paths for the full-chain certificate and key files on the IoT Edge downstream device. IoT Edge requires the certificate and private key to be in text-based privacy-enhanced mail (PEM) format. For example:

```
toml
```

```
[edge_ca]
cert = "file:///var/aziot/certs/iot-device-downstream-full-
chain.cert.pem"
pk = "file:///var/aziot/secrets/iot-device-downstream.key.pem"
```

6. Verify your IoT Edge device uses the correct version of the IoT Edge agent when it starts. Find the **Default Edge Agent** section and set the **image** value for IoT Edge to version 1.5. For example:

```
toml
```

```
[agent]
name = "edgeAgent"
type = "docker"

[agent.config]
image: "mcr.microsoft.com/azureiotedge-agent:1.5"
```

7. The beginning of your downstream configuration file should look similar to the following example.

```
toml

parent_hostname = "10.0.0.4"
trust_bundle_cert = "file:///var/aziot/certs/azure-iot-test-
only.root.ca.cert.pem"

[edge_ca]
cert = "file:///var/aziot/certs/iot-device-downstream-full-
chain.cert.pem"
pk = "file:///var/aziot/secrets/iot-device-downstream.key.pem"
```

8. Save and close the `config.toml` configuration file. For example if you're using the `nano` editor, select **Ctrl+O - Write Out**, **Enter**, and **Ctrl+X - Exit**.
9. If you've used any other certificates for IoT Edge before, delete the files in the following two directories to make sure that your new certificates get applied:

- `/var/lib/aziot/certd/certs`
- `/var/lib/aziot/keyd/keys`

10. Apply your changes.

```
Bash

sudo iotedge config apply
```

11. Check for any errors in the configuration.

```
Bash

sudo iotedge check --verbose
```

 Tip

The IoT Edge check tool uses a container to perform some of the diagnostics check. If you want to use this tool on downstream IoT Edge devices, make sure they can access `mcr.microsoft.com/azureiotedge-diagnostics:latest`, or have the container image in your private container registry.

ⓘ Note

On a newly provisioned device, you may see an error related to IoT Edge Hub:

x production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error

Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module isn't running. To resolve the error, in IoT Hub, set the modules for the device and create a deployment. Creating a deployment for the device starts the modules on the device including the IoT Edge Hub module.

Network isolate downstream devices

The steps so far in this article set up IoT Edge devices as either a gateway or a downstream device, and create a trusted connection between them. The gateway device handles interactions between the downstream device and IoT Hub, including authentication and message routing. Modules deployed to downstream IoT Edge devices can still create their own connections to cloud services.

Some network architectures, like those that follow the ISA-95 standard, seek to minimize the number of internet connections. In those scenarios, you can configure downstream IoT Edge devices without direct internet connectivity. Beyond routing IoT Hub communications through their gateway device, downstream IoT Edge devices can rely on the gateway device for all cloud connections.

This network configuration requires that only the IoT Edge device in the top layer of a gateway hierarchy has direct connections to the cloud. IoT Edge devices in the lower layers can only communicate with their parent device or any children devices. Special modules on the gateway devices enable this scenario, including:

- The **API proxy module** is required on any IoT Edge gateway that has another IoT Edge device below it. That means it must be on *every layer* of a gateway hierarchy except the bottom layer. This module uses an [nginx](#) reverse proxy to route HTTP

data through network layers over a single port. It's highly configurable through its module twin and environment variables, so can be adjusted to fit your gateway scenario requirements.

- The **Docker registry module** can be deployed on the IoT Edge gateway at the *top layer* of a gateway hierarchy. This module is responsible for retrieving and caching container images on behalf of all the IoT Edge devices in lower layers. The alternative to deploying this module at the top layer is to use a local registry, or to manually load container images onto devices and set the module pull policy to **never**.
- The **Azure Blob Storage on IoT Edge** can be deployed on the IoT Edge gateway at the *top layer* of a gateway hierarchy. This module is responsible for uploading blobs on behalf of all the IoT Edge devices in lower layers. The ability to upload blobs also enables useful troubleshooting functionality for IoT Edge devices in lower layers, like module log upload and support bundle upload.

Network configuration

For each gateway device in the top layer, network operators need to:

- Provide a static IP address or fully qualified domain name (FQDN).
- Authorize outbound communications from this IP address to your Azure IoT Hub hostname over ports 443 (HTTPS) and 5671 (AMQP).
- Authorize outbound communications from this IP address to your Azure Container Registry hostname over port 443 (HTTPS).

The API proxy module can only handle connections to one container registry at a time. We recommend having all container images, including the public images provided by Microsoft Container Registry (mcr.microsoft.com), stored in your private container registry.

For each gateway device in a lower layer, network operators need to:

- Provide a static IP address.
- Authorize outbound communications from this IP address to the parent gateway's IP address over ports 443 (HTTPS) and 5671 (AMQP).

Deploy modules to top layer devices

The IoT Edge device at the top layer of a gateway hierarchy has a set of required modules that must be deployed to it, in addition to any workload modules you may run on the device.

The API proxy module was designed to be customized to handle most common gateway scenarios. This article provides an example to set up the modules in a basic configuration. Refer to [Configure the API proxy module for your gateway hierarchy scenario](#) for more detailed information and examples.

Portal

1. In the [Azure portal](#), navigate to your IoT hub.
2. Select **Devices** under the **Device management** menu.
3. Select the top layer IoT Edge device that you're configuring from the list.
4. Select **Set modules**.
5. In the **IoT Edge modules** section, select **Add** then choose **IoT Edge Module**.
6. Update the following module settings:

 Expand table

Setting	Value
IoT Module name	IoTEdgeAPIProxy
Image URI	mcr.microsoft.com/azureiotedge-api-proxy:latest
Restart policy	always
Desired status	running

If you want to use a different version or architecture of the API proxy module, find the available images in the [Microsoft Artifact Registry](#).

- a. In the **Environment variables** tab, add a variable named `NGINX_DEFAULT_PORT` of type *Text* with a value of `443`.
- b. In the **Container create options** tab, update the port bindings to reference port 443.

JSON

```
{  
  "HostConfig": {  
    "PortBindings": {  
      "443/tcp": [  
        {  
          "HostPort": "443"  
        }  
      ]  
    }  
  }  
}
```

These changes configure the API proxy module to listen on port 443. To prevent port binding collisions, you need to configure the edgeHub module to not listen on port 443. Instead, the API proxy module will route any edgeHub traffic on port 443.

7. Select **Add** to add the module to the deployment.
8. Select **Runtime Settings** and find the edgeHub module *Container Create Options*. Delete the port binding for port 443, leaving the bindings for ports 5671 and 8883.

JSON

```
{  
  "HostConfig": {  
    "PortBindings": {  
      "5671/tcp": [  
        {  
          "HostPort": "5671"  
        }  
      ],  
      "8883/tcp": [  
        {  
          "HostPort": "8883"  
        }  
      ]  
    }  
  }  
}
```

9. Select **Apply** to save your changes to the runtime settings.
10. Provide the following values to add the Docker registry module to your deployment.

- a. In the **IoT Edge modules** section, select **Add** then choose **IoT Edge Module**.

[+] Expand table

Setting	Value
IoT Module name	registry
Image URI	registry:latest
Restart policy	always
Desired status	running

- b. In the **Environment variables** tab, add the following variables:

[+] Expand table

Name	Type	Value
REGISTRY_PROXY_REMOTEURL	Text	The URL for the container registry you want this registry module to map to. For example, https://myregistry.azurecr . The registry module can only map to one container registry, so we recommend having all container images in a single private container registry.
REGISTRY_PROXY_USERNAME	Text	Username to authenticate to the container registry.
REGISTRY_PROXY_PASSWORD	Text	Password to authenticate to the container registry.

- c. In the **Container create options** tab, update the port bindings to reference port 5000.

JSON

```
{  
  "HostConfig": {  
    "PortBindings": {  
      "5000/tcp": [  
        {  
          "HostPort": "5000"  
        }  
      ]  
    }  
  }  
}
```

```
}
```

11. Select **Add** to add the module to the deployment.
12. Select **Next: Routes** to go to the next step.
13. To enable device-to-cloud messages from downstream devices to reach IoT Hub, include a route that passes all messages to IoT Hub. For example:
 - a. **Name:** Route
 - b. **Value:** FROM /messages/* INTO \$upstream
14. Select **Review + create** to go to the final step.
15. Select **Create** to deploy to your device.

Deploy modules to lower layer devices

IoT Edge devices in lower layers of a gateway hierarchy have one required module that must be deployed to them, in addition to any workload modules you may run on the device.

Route container image pulls

Before discussing the required proxy module for IoT Edge devices in gateway hierarchies, it's important to understand how IoT Edge devices in lower layers get their module images.

If your lower layer devices can't connect to the cloud, but you want them to pull module images as usual, then the top layer device of the gateway hierarchy must be configured to handle these requests. The top layer device needs to run a Docker **registry** module that is mapped to your container registry. Then, configure the API proxy module to route container requests to it. Those details are discussed in the earlier sections of this article. In this configuration, the lower layer devices shouldn't point to cloud container registries, but to the registry running in the top layer.

For example, instead of calling `mcr.microsoft.com/azureiotedge-api-proxy:1.1`, lower layer devices should call `$upstream:443/azureiotedge-api-proxy:1.1`.

The **\$upstream** parameter points to the parent of a lower layer device, so the request will route through all the layers until it reaches the top layer which has a proxy

environment routing container requests to the registry module. The :443 port in this example should be replaced with whichever port the API proxy module on the parent device is listening on.

The API proxy module can only route to one registry module, and each registry module can only map to one container registry. Therefore, any images that lower layer devices need to pull must be stored in a single container registry.

If you don't want lower layer devices making module pull requests through a gateway hierarchy, another option is to manage a local registry solution. Or, push the module images onto the devices before creating deployments and then set the **imagePullPolicy** to **never**.

Bootstrap the IoT Edge agent

The IoT Edge agent is the first runtime component to start on any IoT Edge device. You need to make sure that any downstream IoT Edge devices can access the `edgeAgent` module image when they start up, and then they can access deployments and start the rest of the module images.

When you go into the config file on an IoT Edge device to provide its authentication information, certificates, and parent hostname, also update the `edgeAgent` container image.

If the top level gateway device is configured to handle container image requests, replace `mcr.microsoft.com` with the parent hostname and API proxy listening port. In the deployment manifest, you can use `$upstream` as a shortcut, but that requires the `edgeHub` module to handle routing and that module hasn't started at this point. For example:

```
toml

[agent]
name = "edgeAgent"
type = "docker"

[agent.config]
image: "{Parent FQDN or IP}:443/azureiotedge-agent:1.5"
```

If you are using a local container registry, or providing the container images manually on the device, update the config file accordingly.

Configure runtime and deploy proxy module

The API proxy module is required for routing all communications between the cloud and any downstream IoT Edge devices. An IoT Edge device in the bottom layer of the hierarchy, with no downstream IoT Edge devices, does not need this module.

The API proxy module was designed to be customized to handle most common gateway scenarios. This article briefly touches on the steps to set up the modules in a basic configuration. Refer to [Configure the API proxy module for your gateway hierarchy scenario](#) for more detailed information and examples.

1. In the [Azure portal](#), navigate to your IoT hub.
2. Select **Devices** under the **Device management** menu.
3. Select the lower layer IoT Edge device that you're configuring from the list.
4. Select **Set modules**.
5. In the **IoT Edge modules** section, select **Add** then choose **IoT Edge Module**.
6. Update the following module settings:

[] [Expand table](#)

Setting	Value
IoT Module name	IoTEdgeAPIProxy
Image URI	mcr.microsoft.com/azureiotedge-api-proxy:latest
Restart policy	always
Desired status	running

If you want to use a different version or architecture of the API proxy module, find the available images in the [Microsoft Artifact Registry](#).

- a. In the **Environment variables** tab, add a variable named `NGINX_DEFAULT_PORT` of type *Text* with a value of `443`.
- b. In the **Container create options** tab, update the port bindings to reference port `443`.

JSON

```
{  
  "HostConfig": {  
    "PortBindings": {  
      "443/tcp": [  
        {"HostPort": "443"}  
      ]  
    }  
  }  
}
```

```
        {
          "HostPort": "443"
        }
      ]
    }
}
```

These changes configure the API proxy module to listen on port 443. To prevent port binding collisions, you need to configure the edgeHub module to not listen on port 443. Instead, the API proxy module will route any edgeHub traffic on port 443.

7. Select **Runtime Settings**.

8. Update the edgeHub module settings:

- In the **Image** field, replace `mcr.microsoft.com` with `$upstream:443`.
- In the **Create options** field, delete the port binding for port 443, leaving the bindings for ports 5671 and 8883.

JSON

```
{
  "HostConfig": {
    "PortBindings": {
      "5671/tcp": [
        {
          "HostPort": "5671"
        }
      ],
      "8883/tcp": [
        {
          "HostPort": "8883"
        }
      ]
    }
  }
}
```

9. Update the edgeAgent module settings:

- In the **Image** field, replace `mcr.microsoft.com` with `$upstream:443`.

10. Select **Apply** to save your changes to the runtime settings.

11. Select **Next: Routes** to go to the next step.

12. To enable device-to-cloud messages from downstream devices to reach IoT Hub, include a route that passes all messages to `$upstream`. The upstream parameter

points to the parent device in the case of lower layer devices. For example:

- a. **Name:** Route
- b. **Value:** FROM /messages/* INTO \$upstream

13. Select **Review + create** to go to the final step.

14. Select **Create** to deploy to your device.

Verify connectivity from child to parent

1. Verify the TLS/SSL connection from the child to the parent by running the following `openssl` command on the downstream device. Replace `<parent hostname>` with the FQDN or IP address of the parent.

Bash

```
openssl s_client -connect <parent hostname>:8883 </dev/null 2>&1
>/dev/null
```

The command should assert successful validation of the parent certificate chain similar to the following example:

Output

```
azureUser@child-vm:~$ openssl s_client -connect <parent hostname>:8883
</dev/null 2>&1 >/dev/null
Can't use SSL_get_servername
depth=3 CN = Azure_IoT_Hub_CA_Cert_Test_Only
verify return:1
depth=2 CN = Azure_IoT_Hub_Intermediate_Cert_Test_Only
verify return:1
depth=1 CN = gateway.ca
verify return:1
depth=0 CN = <parent hostname>
verify return:1
DONE
```

The "Can't use SSL_get_servername" message can be ignored.

The `depth=0 CN =` value should match the **hostname** parameter specified in the parent's `config.toml` configuration file.

If the command times out, there may be blocked ports between the child and parent devices. Review the network configuration and settings for the devices.

Warning

Not using a full-chain certificate in the gateway's [edge_ca] section results in certificate validation errors from the downstream device. For example, the `openssl s_client ...` command above will produce:

```
Can't use SSL_get_servername
depth=1 CN = gateway.ca
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 CN = <parent hostname>
verify return:1
DONE
```

The same issue occurs for TLS-enabled devices that connect to the downstream IoT Edge device if the full-chain device certificate isn't used and configured on the downstream device.

Integrate Microsoft Defender for IoT with IoT Edge gateway

Downstream devices can be used to integrate the Microsoft Defender for IoT's micro agent with the IoT Edge gateway using downstream device proxying.

Learn more about the [Defender for IoT micro agent](#).

To integrate Microsoft Defender for IoT with IoT Edge using downstream device proxying:

1. Sign in to the Azure portal.
2. Navigate to **IoT Hub** > **Your Hub** > **Device management** > **Devices**
3. Select your device.

Microsoft Azure (Preview) Report a bug Search resources, services, and docs (G+)

Home > IoT Hub > Kfir1

IoT Hub

Microsoft (microsoft.onmicrosoft.com)

Create Manage view ...

kfir1

Name ↑↓

Kfir1 ...

Device management

- Devices
- IoT Edge
- Configurations
- Updates
- Queries

Hub settings

- Built-in endpoints
- Message routing

Kfir1 | Devices

IoT Hub

Search (Ctrl+/)

View, create, delete, and update devices in your IoT hub

Device name

enter device ID

Find devices

Add Device Refresh Delete

Device ID

dev1

shmulik1

PackageTest1

test2

422

leaf1

StressTest-Debian

shomorod1

4. Select the `DefenderIotMicroAgent` module twin that you created from [these instructions](#).

[Home](#) > [IoT Hub](#) > [Kfir1](#) >

leaf1



...

Kfir1



Save



Message to Device



Direct Method



Add Module Id

Device ID



leaf1

Primary Key



.....

Secondary Key



.....

Primary Connection String



.....

Secondary Connection String



.....

Enable connection to IoT Hub



Enable



Disable

Parent device



No parent device

[Module Identities](#)[Configurations](#)

Module ID

Connection State

DefenderIotMicroAgent

Disconnected

5. Select the button to copy your Connection string (primary key).

6. Paste the connection string into a text editing application, and add the **GatewayHostName** to the string. The **GatewayHostName** is the fully qualified domain name or IP address of the parent device. For example,

HostName=nested11.azure-

```
devices.net;DeviceId=downstream1;ModuleId=module1;SharedAccessKey=xxx;GatewayHost  
Name=10.16.7.4.
```

7. Open a terminal on the downstream device.
8. Use the following command to place the connection string encoded in utf-8 in the Defender for Cloud agent directory into the file `connection_string.txt` in the following path: `/etc/defender_iot_micro_agent/connection_string.txt`:

Bash

```
sudo bash -c 'echo "<connection string>" >  
/etc/defender_iot_micro_agent/connection_string.txt'
```

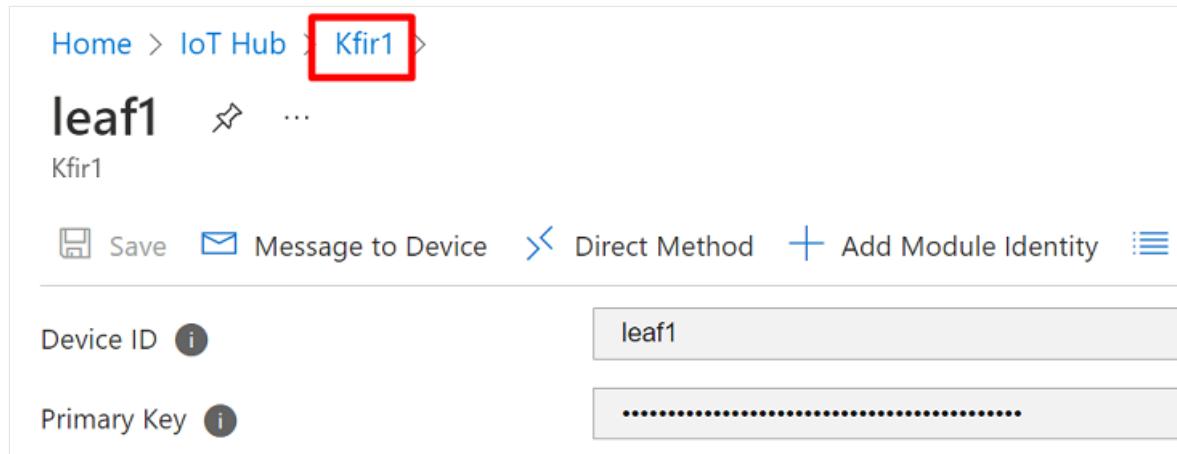
The `connection_string.txt` should now be located in the following path location `/etc/defender_iot_micro_agent/connection_string.txt`.

9. Restart the service using this command:

Bash

```
sudo systemctl restart defender-iot-micro-agent.service
```

10. Navigate back to the device.



11. Enable the connection to the IoT Hub, and select the gear icon.

leaf1

Kfir1

Device ID: leaf1

Primary Key:
Secondary Key:

Primary Connection String:
Secondary Connection String:

Enable connection to IoT Hub: Enable Disable

Parent device: No parent device

Module Identities Configurations

12. Select the parent device from the displayed list.
13. Ensure that port 8883 (MQTT) between the downstream device and the IoT Edge device is open.

Next steps

[How an IoT Edge device can be used as a gateway](#)

[Configure the API proxy module for your gateway hierarchy scenario](#)

Feedback

Was this page helpful?

Yes

No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Configure the API proxy module for your gateway hierarchy scenario

Article • 06/12/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article walks through the configuration options for the API proxy module, so you can customize the module to support your gateway hierarchy requirements.

The API proxy module simplifies communication for IoT Edge devices when multiple services are deployed that all support HTTPS protocol and bind to port 443. This is especially relevant in hierarchical deployments of IoT Edge devices in ISA-95-based network-isolated architectures like those described in [Network isolate downstream devices](#) because the clients on the downstream devices can't connect directly to the cloud.

For example, to allow downstream IoT Edge devices to pull Docker images requires deploying a Docker registry module. To allow uploading blobs requires deploying an Azure Blob Storage module on the same IoT Edge device. Both these services use HTTPS for communication. The API proxy enables such deployments on an IoT Edge device. Instead of each service, the API proxy module binds to port 443 on the host device and routes the request to the correct service module running on that device per user-configurable rules. The individual services are still responsible for handling the requests, including authenticating and authorizing the clients.

Without the API proxy, each service module would have to bind to a separate port on the host device, requiring a tedious and error-prone configuration change on each child device that connects to the parent IoT Edge device.

Note

A downstream device emits data directly to the Internet or to gateway devices (IoT Edge-enabled or not). A child device can be a downstream device or a gateway device in a nested topology.

Deploy the proxy module

The API proxy module is available from the [Microsoft Container Registry \(MCR\)](#) and the image URI is `mcr.microsoft.com/azureiotedge-api-proxy:latest`. You can deploy the module using the [Azure portal](#) or [Azure CLI](#).

Understand the proxy module

The API proxy module leverages an nginx reverse proxy to route data through network layers. A proxy is embedded in the module, which means that the module image needs to support the proxy configuration. For example, if the proxy is listening on a certain port then the module needs to have that port open.

The proxy starts with a default configuration file embedded in the module. You can pass a new configuration to the module from the cloud using its [module twin](#). Additionally, you can use environment variables to turn configuration settings on or off at deployment time.

This article focuses first on the default configuration file, and how to use environment variables to enable its settings. Then, we discuss customizing the configuration file at the end.

Default configuration

The API proxy module comes with a default configuration that supports common scenarios and allows for customization. You can control the default configuration through environment variables of the module.

Currently, the default environment variables include:

[+] Expand table

Environment variable	Description
<code>PROXY_CONFIG_ENV_VAR_LIST</code>	List all the variables that you intend to update in a comma-separated list. This step prevents accidentally modifying the wrong configuration settings.

Environment variable	Description
<code>NGINX_DEFAULT_TLS</code>	<p>Specifies the list of TLS protocols to be enabled. See NGINX's ssl_protocols.</p> <p>Default is 'TLSv1.2'.</p>
<code>NGINX_DEFAULT_PORT</code>	<p>Changes the port that the nginx proxy listens to. If you update this environment variable, you must expose the port in the module dockerfile and declare the port binding in the deployment manifest. For more information, see Expose proxy port.</p> <p>Default is 443.</p> <p>When deployed from the Azure Marketplace, the default port is updated to 8000, to prevent conflicts with the edgeHub module. For more information, see Minimize open ports.</p>
<code>DOCKER_REQUEST_ROUTE_ADDRESS</code>	<p>Address to route docker requests. Modify this variable on the top layer device to point to the registry module.</p> <p>Default is the parent hostname.</p>
<code>BLOB_UPLOAD_ROUTE_ADDRESS</code>	<p>Address to route blob registry requests. Modify this variable on the top layer device to point to the blob storage module.</p> <p>Default is the parent hostname.</p>

Minimize open ports

To minimize the number of open ports, the API proxy module should relay all HTTPS traffic (port 443), including traffic targeting the edgeHub module. The API proxy module is configured by default to re-route all edgeHub traffic on port 443.

Use the following steps to configure your deployment to minimize open ports:

1. Update the edgeHub module settings to not bind on port 443, otherwise there will be port binding conflicts. By default, the edgeHub module binds on ports 443, 5671, and 8883. Delete the port 443 binding and leave the other two in place:

JSON

```
{
  "HostConfig": {
    "PortBindings": {
      "5671/tcp": [
        {
          "HostPort": "5671"
        }
      ]
    }
  }
}
```

```
        "HostPort": "5671"
    }
],
"8883/tcp": [
{
    "HostPort": "8883"
}
]
}
}
```

2. Configure the API proxy module to bind on port 443.

- a. Set value of the **NGINX_DEFAULT_PORT** environment variable to `443`.
- b. Update the container create options to bind on port 443.

JSON

```
{
  "HostConfig": {
    "PortBindings": {
      "443/tcp": [
        {
          "HostPort": "443"
        }
      ]
    }
  }
}
```

If you don't need to minimize open ports, then you can let the edgeHub module use port 443 and configure the API proxy module to listen on another port. For example, the API proxy module can listen on port 8000 by setting the value of the **NGINX_DEFAULT_PORT** environment variable to `8000` and creating a port binding for port 8000.

Enable container image download

A common use case for the API proxy module is to enable IoT Edge devices in lower layers to pull container images. This scenario uses the [Docker registry module](#) to retrieve container images from the cloud and cache them at the top layer. The API proxy relays all HTTPS requests to download a container image from the lower layers to be served by the registry module in the top layer.

This scenario requires that downstream IoT Edge devices point to the domain name `$upstream` followed by the API Proxy module port number instead of the container registry of an image. For example: `$upstream:8000/azureiotedge-api-proxy:1.1`.

This use case is demonstrated in the tutorial [Create a hierarchy of IoT Edge devices using gateways](#).

Configure the following modules at the **top layer**:

- A Docker registry module
 - Configure the module with a memorable name like *registry* and expose a port in the module to receive requests.
 - Configure the module to map to your container registry.
- An API proxy module
 - Configure the following environment variables:

[+] [Expand table](#)

Name	Value
<code>DOCKER_REQUEST_ROUTE_ADDRESS</code>	The registry module name and open port. For example, <code>registry:5000</code> .
<code>NGINX_DEFAULT_PORT</code>	The port that the nginx proxy listens on for requests from downstream devices. For example, <code>8000</code> .

- Configure the following `createOptions`:

JSON

```
{  
    "HostConfig": {  
        "PortBindings": {  
            "8000/tcp": [  
                {  
                    "HostPort": "8000"  
                }  
            ]  
        }  
    }  
}
```

Configure the following module on any **lower layer** for this scenario:

- API proxy module. The API proxy module is required on all lower layer devices except the bottom layer device.

- Configure the following environment variables:

[Expand table](#)

Name	Value
NGINX_DEFAULT_PORT	The port that the nginx proxy listens on for requests from downstream devices. For example, 8000.

- Configure the following createOptions:

JSON

```
{
  "HostConfig": {
    "PortBindings": {
      "8000/tcp": [
        {
          "HostPort": "8000"
        }
      ]
    }
  }
}
```

Expose proxy port

Port 8000 is exposed by default from the docker image. If a different nginx proxy port is used, add the **ExposedPorts** section declaring the port in the deployment manifest. For example, if you change the nginx proxy port to 8001, add the following to the deployment manifest:

JSON

```
{
  "ExposedPorts": {
    "8001/tcp": {}
  },
  "HostConfig": {
    "PortBindings": {
      "8001/tcp": [
        {
          "HostPort": "8001"
        }
      ]
    }
  }
}
```

```
    }  
}
```

Enable blob upload

Another use case for the API proxy module is to enable IoT Edge devices in lower layers to upload blobs. This use case enables troubleshooting functionality on lower layer devices like uploading module logs or uploading the support bundle.

This scenario uses the [Azure Blob Storage on IoT Edge](#) module at the top layer to handle blob creation and upload. In a nested scenario, up to five layers are supported. The *Azure Blob Storage on IoT Edge* module is required on the top layer device and optional for lower layer devices. For a sample multi-layer deployment, see the [Azure IoT Edge for Industrial IoT](#) sample.

Configure the following modules at the **top layer**:

- An Azure Blob Storage on IoT Edge module.
- An API proxy module
 - Configure the following environment variables:

[\[\] Expand table](#)

Name	Value
BLOB_UPLOAD_ROUTE_ADDRESS	The blob storage module name and open port. For example, <code>azureblobstorageoniotedge:11002</code> .
NGINX_DEFAULT_PORT	The port that the nginx proxy listens on for requests from downstream devices. For example, <code>8000</code> .

- Configure the following createOptions:

JSON

```
{  
  "HostConfig": {  
    "PortBindings": {  
      "8000/tcp": [  
        {  
          "HostPort": "8000"  
        }  
      ]  
    }  
  }  
}
```

```
    }  
}
```

Configure the following module on any **lower layer** for this scenario:

- An API proxy module
 - Configure the following environment variables:

[\[+\] Expand table](#)

Name	Value
NGINX_DEFAULT_PORT	The port that the nginx proxy listens on for requests from downstream devices. For example, <code>8000</code> .

- Configure the following `createOptions`:

JSON

```
{  
  "HostConfig": {  
    "PortBindings": {  
      "8000/tcp": [  
        {  
          "HostPort": "8000"  
        }  
      ]  
    }  
  }  
}
```

Use the following steps to upload the support bundle or log file to the blob storage module located at the top layer:

1. Create a blob container, using either Azure Storage Explorer or the REST APIs. For more information, see [Store data at the edge with Azure Blob Storage on IoT Edge](#).
2. Request a log or support bundle upload according to the steps in [Retrieve logs from IoT Edge deployments](#), but use the domain name `$upstream` and the open proxy port in place of the blob storage module address. For example:

JSON

```
{  
  "schemaVersion": "1.0",  
  "sasUrl": "https://$upstream:8000/myBlobStorageName/myContainerName?  
SAS_key",
```

```
        "since": "2d",
        "until": "1d",
        "edgeRuntimeOnly": false
    }
```

Edit the proxy configuration

A default configuration file is embedded in the API proxy module, but you can pass a new configuration to the module via the cloud using the module twin.

When you write your own configuration, you can still use environment to adjust settings per deployment. Use the following syntax:

- Use `#{MY_ENVIRONMENT_VARIABLE}` to retrieve the value of an environment variable.
- Use conditional statements to turn settings on or off based on the value of an environment variable:

```
conf
```

```
#if_tag ${MY_ENVIRONMENT_VARIABLE}
    statement to execute if environment variable evaluates to 1
#endif_tag ${MY_ENVIRONMENT_VARIABLE}

#if_tag !${MY_ENVIRONMENT_VARIABLE}
    statement to execute if environment variable evaluates to 0
#endif_tag !${MY_ENVIRONMENT_VARIABLE}
```

When the API proxy module parses a proxy configuration, it first replaces all environment variables listed in the `PROXY_CONFIG_ENV_VAR_LIST` with their provided values using substitution. Then, everything between an `#if_tag` and `#endif_tag` pair is replaced. The module then provides the parsed configuration to the nginx reverse proxy.

To update the proxy configuration dynamically, use the following steps:

1. Write your configuration file. You can use this default template as a reference:
[nginx_default_config.conf](#)
2. Copy the text of the configuration file and convert it to base64.
3. Paste the encoded configuration file as the value of the `proxy_config` desired property in the module twin.

Update IoT Edge Module

Specify the settings for an IoT Edge custom module.

[Learn more](#)

IoT Edge Module Name *

ApiProxyModule

[Module Settings](#) [Environment Variables](#) [Container Create Options](#) [Module Twin Settings](#)

Module twin desired properties will be updated to reflect entered JSON.

[Learn more about deployments](#)

```
1  {
2    "proxy_config": "ewogICJkZXZpY2VBdXRvRGVsZXRlUHJvcGVydGllcyI6IHsKICAgI
3  }
```

Next steps

Use the API proxy module to [Connect a downstream IoT Edge device to an Azure IoT Edge gateway](#).

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#) ↗

Store data at the edge with Azure Blob Storage on IoT Edge

Article • 06/06/2024

Applies to: IoT Edge 1.5 IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure Blob Storage on IoT Edge provides a [block blob](#) and [append blob](#) storage solution at the edge. A blob storage module on your IoT Edge device behaves like an Azure blob service, except the blobs are stored locally on your IoT Edge device. You can access your blobs using the same Azure storage SDK methods or blob API calls that you're already used to. This article explains the concepts related to Azure Blob Storage on IoT Edge container that runs a blob service on your IoT Edge device.

This module is useful in scenarios:

- Where data needs to be stored locally until it can be processed or transferred to the cloud. This data can be videos, images, finance data, hospital data, or any other unstructured data.
- When devices are located in a place with limited connectivity.
- When you want to efficiently process the data locally to get low latency access to the data, such that you can respond to emergencies as quickly as possible.
- When you want to reduce bandwidth costs and avoid transferring terabytes of data to the cloud. You can process the data locally and send only the processed data to the cloud.

This module comes with `deviceToCloudUpload` and `deviceAutoDelete` features.

The `deviceToCloudUpload` feature is a configurable functionality. This function automatically uploads the data from your local blob storage to Azure with intermittent internet connectivity support. It allows you to:

- Turn ON/OFF the `deviceToCloudUpload` feature.
- Choose the order in which the data is copied to Azure like `NewestFirst` or `OldestFirst`.
- Specify the Azure Storage account to which you want your data uploaded.
- Specify the containers you want to upload to Azure. This module allows you to specify both source and target container names.
- Choose the ability to delete the blobs immediately, after upload to cloud storage is finished
- Do full blob upload (using `Put Blob` operation) and block level upload (using `Put Block`, `Put Block List` and `Append Block` operations).

This module uses block level upload, when your blob consists of blocks. Here are some of the common scenarios:

- Your application updates some blocks of a previously uploaded block blob or appends new blocks to an append blob. This module uploads only the updated blocks and not the whole blob.
- The module is uploading blob and internet connection goes away, when the connectivity is back again it uploads only the remaining blocks and not the whole blob.

If an unexpected process termination (like power failure) happens during a blob upload, all blocks due for the upload are uploaded again once the module comes back online.

deviceAutoDelete is a configurable functionality. This function automatically deletes your blobs from the local storage when the specified duration (measured in minutes) expires. It allows you to:

- Turn ON/OFF the deviceAutoDelete feature.
- Specify the time in minutes (deleteAfterMinutes) after which the blobs are automatically deleted.
- Choose the ability to retain the blob while it's uploading if the deleteAfterMinutes value expires.

Prerequisites

An Azure IoT Edge device:

- You can use your development machine or a virtual machine as an IoT Edge device by following the steps in the quickstart for [Linux](#) or [Windows devices](#).
- Refer to [Azure IoT Edge supported systems](#) for a list of supported operating systems and architectures. The Azure Blob Storage on IoT Edge module supports following architectures:
 - Windows AMD64
 - Linux AMD64
 - Linux ARM32
 - Linux ARM64

Cloud resources:

A standard-tier [IoT Hub](#) in Azure.

deviceToCloudUpload and deviceAutoDelete properties

Use the module's desired properties to set **deviceToCloudUploadProperties** and **deviceAutoDeleteProperties**. Desired properties can be set during deployment or changed later by editing the module twin without the need to redeploy. We recommend checking the "Module Twin" for `reported configuration` and `configurationValidation` to make sure values are correctly propagated.

deviceToCloudUploadProperties

The name of this setting is `deviceToCloudUploadProperties`. If you're using the IoT Edge simulator, set the values to the related environment variables for these properties, which you can find in the explanation section.

 Expand table

Property	Possible Values	Explanation
uploadOn	true, false	Set to <code>false</code> by default. If you want to turn on the feature, set this field to <code>true</code> . Environment variable: <code>deviceToCloudUploadProperties__uploadOn={false,true}</code>
uploadOrder	NewestFirst, OldestFirst	Allows you to choose the order in which the data is copied to Azure. Set to <code>OldestFirst</code> by default. The order is determined by last modified time of blob. Environment variable: <code>deviceToCloudUploadProperties__uploadOrder={NewestFirst,OldestFirst}</code>
cloudStorageConnectionString		<code>"DefaultEndpointsProtocol=https;AccountName=<your Azure Storage Account Name>;AccountKey=<your Azure Storage Account Key>;EndpointSuffix=<your end point</code>

Property	Possible Values	Explanation
		<p><code><suffix></code> is a connection string that allows you to specify the storage account to which you want your data uploaded. Specify <code>Azure Storage Account Name</code>, <code>Azure Storage Account Key</code>, <code>End point suffix</code>. Add appropriate EndpointSuffix of Azure where data is uploaded, it varies for Global Azure, Government Azure, and Microsoft Azure Stack.</p> <p>You can choose to specify Azure Storage SAS connection string here. But you have to update this property when it expires. SAS permissions may include create access for containers and create, write, and add access for blobs.</p> <p>Environment variable: <code>deviceToCloudUploadProperties__cloudStorageConnectionString=<connection string></code></p>
storageContainersForUpload	<pre>"<source container name1>: { "target": "<target container name>", "<source container name1>: { "target": "%h-%d-%m- %c", } "<source container name1>: { "target": "%d-%c" }</pre>	<p>Allows you to specify the container names you want to upload to Azure. This module allows you to specify both source and target container names. If you don't specify the target container name, it's automatically assigned a container name such as <code><IoTHubName>-<IoTEdgeDeviceID>-<ModuleName>-<SourceContainerName></code>. You can create template strings for target container name, check out the possible values column.</p> <ul style="list-style-type: none"> * %h -> IoT Hub Name (3-50 characters). * %d -> IoT Edge Device ID (1 to 129 characters). * %m -> Module Name (1 to 64 characters). * %c -> Source Container Name (3 to 63 characters). <p>Maximum size of the container name is 63 characters. The name is automatically assigned the target container name if the size of container exceeds 63 characters. In this case, name is trimmed in each section (IoTHubName, IoTEdgeDeviceID, ModuleName, SourceContainerName) to 15 characters.</p> <p>Environment variable: <code>deviceToCloudUploadProperties__storageContainersForUpload__<sourceName>__target=<targetName></code></p>
deleteAfterUpload	true, false	<p>Set to <code>false</code> by default. When set to <code>true</code>, the data automatically deletes when the upload to cloud storage is finished.</p> <p>CAUTION: If you're using append blobs, this setting deletes append blobs from local storage after a successful upload, and any future Append Block operations to those blobs will fail. Use this setting with caution. Don't enable this setting if your application does infrequent append operations or doesn't support continuous append operations</p> <p>Environment variable: <code>deviceToCloudUploadProperties__deleteAfterUpload={false,true}</code>.</p>

deviceAutoDeleteProperties

The name of this setting is `deviceAutoDeleteProperties`. If you're using the IoT Edge simulator, set the values to the related environment variables for these properties, which you can find in the explanation section.

 Expand table

Property	Possible Values	Explanation
deleteOn	true, false	<p>Set to <code>false</code> by default. If you want to turn on the feature, set this field to <code>true</code>.</p> <p>Environment variable: <code>deviceAutoDeleteProperties__deleteOn={false,true}</code></p>
deleteAfterMinutes	<code><minutes></code>	<p>Specify the time in minutes. The module automatically deletes your blobs from local storage when this value expires. Current maximum minutes allowed are 35791.</p> <p>Environment variable: <code>deviceAutoDeleteProperties__deleteAfterMinutes=<minutes></code></p>
retainWhileUploading	true, false	<p>By default it's set to <code>true</code>, and retains the blob while it's uploading to cloud storage if <code>deleteAfterMinutes</code> expire. You can set it to <code>false</code> and it deletes the data as soon as <code>deleteAfterMinutes</code> expires. Note: For this property to work <code>uploadOn</code> should be set to <code>true</code>.</p> <p>CAUTION: If you use append blobs, this setting deletes append blobs from local storage when the value expires, and any future Append Block operations to those blobs fail. Make sure the expiry value is large enough for the expected frequency of append operations performed by your application.</p> <p>Environment variable: <code>deviceAutoDeleteProperties__retainWhileUploading={false,true}</code></p>

Using SMB share as your local storage

You can provide SMB share as your local storage path, when you deploy Windows container of this module on Windows host.

Make sure the SMB share and IoT device are in mutually trusted domains.

You can run `New-SmbGlobalMapping` PowerShell command to map the SMB share locally on the IoT device running Windows.

The configuration steps:

PowerShell

```
$creds = Get-Credential
New-SmbGlobalMapping -RemotePath <remote SMB path> -Credential $creds -LocalPath <Any available
drive letter>
```

For example:

PowerShell

```
$creds = Get-Credential
New-SmbGlobalMapping -RemotePath \\contosofileservice\share1 -Credential $creds -LocalPath G:
```

This command uses the credentials to authenticate with the remote SMB server. Then, map the remote share path to G: drive letter (can be any other available drive letter). The IoT device now has the data volume mapped to a path on the G: drive.

Make sure the user in IoT device can read/write to the remote SMB share.

For your deployment the value of `<storage_mount>` can be `G:/ContainerData:C:/BlobRoot`.

Granting directory access to container user on Linux

If you use [volume mount](#) for storage in your create options for Linux containers, then you don't have to do any extra steps, but if you use [bind mount](#), then these steps are required to run the service correctly.

Following the principle of least privilege to limit the access rights for users to bare minimum permissions they need to perform their work, this module includes a user (name: absie, ID: 11000) and a user group (name: absie, ID: 11000). If the container is started as **root** (default user is **root**), our service is started as the low-privilege **absie** user.

This behavior makes configuration of the permissions on host path binds crucial for the service to work correctly, otherwise the service crashes with access denied errors. The path that is used in directory binding needs to be accessible by the container user (example: absie 11000). You can grant the container user access to the directory by executing these commands on the host:

terminal

```
sudo chown -R 11000:11000 <blob-dir>
sudo chmod -R 700 <blob-dir>
```

For example:

terminal

```
sudo chown -R 11000:11000 /srv/containerdata
sudo chmod -R 700 /srv/containerdata
```

If you need to run the service as a user other than **absie**, you can specify your custom user ID in `createOptions` under "User" property in your deployment manifest. In such a case, use default or root group ID `0`.

JSON

```
"createOptions": {
  "User": "<custom user ID>:0"
}
```

Now, grant the container user access to the directory

terminal

```
sudo chown -R <user ID>:<group ID> <blob-dir>
sudo chmod -R 700 <blob-dir>
```

Configure log files

The default output log level is 'Info'. To change the output log level, set the `LogLevel` environment variable for this module in the deployment manifest. `LogLevel` accepts the following values:

- Critical
- Error
- Warning
- Info
- Debug

For information on configuring log files for your module, see these [production best practices](#).

Connect to your blob storage module

You can use the account name and account key that you configured for your module to access the blob storage on your IoT Edge device.

Specify your IoT Edge device as the blob endpoint for any storage requests that you make to it. You can [Create a connection string for an explicit storage endpoint](#) using the IoT Edge device information and the account name that you configured.

- For modules that are deployed on the same device as where the Azure Blob Storage on IoT Edge module is running, the blob endpoint is: `http://<module name>:11002/<account name>`.
- For modules or applications running on a different device, you have to choose the right endpoint for your network. Depending on your network setup, choose an endpoint format such that the data traffic from your external module or application can reach the device running the Azure Blob Storage on IoT Edge module. The blob endpoint for this scenario is one of:
 - `http://<device IP >:11002/<account name>`
 - `http://<IoT Edge device hostname>:11002/<account name>`
 - `http://<fully qualified domain name>:11002/<account name>`

Important

Azure IoT Edge is case-sensitive when you make calls to modules, and the Storage SDK also defaults to lowercase. Although the name of the module in the [Azure Marketplace](#) is **AzureBlobStorageonIoTEdge**, changing the name to lowercase helps to ensure that your connections to the Azure Blob Storage on IoT Edge module aren't interrupted.

Azure Blob Storage quickstart samples

The Azure Blob Storage documentation includes quickstart sample code in several languages. You can run these samples to test Azure Blob Storage on IoT Edge by changing the blob endpoint to connect to your local blob storage module.

The following quickstart samples use languages that are also supported by IoT Edge, so you could deploy them as IoT Edge modules alongside the blob storage module:

- [.NET](#)
 - The Azure Blob Storage on IoT Edge module v1.4.0 and earlier are compatible with WindowsAzure.Storage 9.3.3 SDK and v1.4.1 also supports Azure.Storage.Blobs 12.8.0 SDK.
- [Python](#)
 - Versions before V2.1 of the Python SDK have a known issue where the module doesn't return the blob creation time. Because of that issue, some methods like list blobs don't work. As a workaround, explicitly set the API version on the blob client to '2017-04-17'. Example: `block_blob_service._X_MS_VERSION = '2017-04-17'`
 - [Append Blob Sample](#)
- [Node.js](#)
- [JS/HTML](#)
- [Ruby](#)
- [Go](#)

- PHP

Connect to your local storage with Azure Storage Explorer

You can use [Azure Storage Explorer](#) to connect to your local storage account.

1. Download and install Azure Storage Explorer
2. The latest version of Azure Storage Explorer uses a newer storage API version not supported by the blob storage module. Start Azure Storage Explorer. Select the **Edit** menu. Verify the **Target Azure Stack Hub APIs** is selected. If it isn't, select **Target Azure Stack Hub**. Restart Azure Storage Explorer for the change to take effect. This configuration is required for compatibility with your IoT Edge environment.
3. Connect to Azure Storage using a connection string
4. Provide connection string: `DefaultEndpointsProtocol=http;BlobEndpoint=http://<host device name>:11002/<your local account name>;AccountName=<your local account name>;AccountKey=<your local account key>;`
5. Go through the steps to connect.
6. Create container inside your local storage account
7. Start uploading files as Block blobs or Append Blobs.

(!) Note

This module does not support Page blobs.

8. You can choose to connect your Azure storage accounts in Storage Explorer, too. This configuration gives you a single view for both your local storage account and Azure storage account

Supported storage operations

Blob storage modules on IoT Edge use the Azure Storage SDKs, and are consistent with the 2017-04-17 version of the Azure Storage API for block blob endpoints.

Because not all Azure Blob Storage operations are supported by Azure Blob Storage on IoT Edge, this section lists the status of each.

Account

Supported:

- List containers

Unsupported:

- Get and set blob service properties
- Preflight blob request
- Get blob service stats

- Get account information

Containers

Supported:

- Create and delete container
- Get container properties and metadata
- List blobs
- Get and set container ACL
- Set container metadata

Unsupported:

- Lease container

Blobs

Supported:

- Put, get, and delete blob
- Get and set blob properties
- Get and set blob metadata

Unsupported:

- Lease blob
- Snapshot blob
- Copy and abort copy blob
- Undelete blob
- Set blob tier

Block blobs

Supported:

- Put block
- Put and get blocklist

Unsupported:

- Put block from URL

Append blobs

Supported:

- Append block

Unsupported:

- Append block from URL

Event Grid on IoT Edge Integration

⊗ Caution

The integration with Event Grid on IoT Edge is in preview

This Azure Blob Storage on IoT Edge module now provides integration with Event Grid on IoT Edge. For detailed information on this integration, see the [tutorial to deploy the modules, publish events and verify event delivery](#).

Release Notes

Here are the [release notes in docker hub](#) for this module. You might be able to find more information related to bug fixes and remediation in the release notes of a specific version.

Next steps

Learn how to [Deploy Azure Blob Storage on IoT Edge](#)

Stay up-to-date with recent updates and announcement on the [Azure Blob Storage on IoT Edge release notes](#) page.

Deploy the Azure Blob Storage on IoT Edge module to your device

Article • 08/12/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

There are several ways to deploy modules to an IoT Edge device and all of them work for Azure Blob Storage on IoT Edge modules. The two simplest methods are to use the Azure portal or Visual Studio Code templates.

Prerequisites

- An [IoT hub](#) in your Azure subscription.
- An IoT Edge device.

If you don't have an IoT Edge device set up, you can create one in an Azure virtual machine. Follow the steps in one of the quickstart articles to [Create a virtual Linux device](#) or [Create a virtual Windows device](#).

- [Visual Studio Code](#).
- [Azure IoT Edge](#) extension. The *Azure IoT Edge tools for Visual Studio Code* extension is in [maintenance mode](#).
- [Azure IoT Hub](#) extension if deploying from Visual Studio Code.

Deploy from the Azure portal

The Azure portal guides you through creating a deployment manifest and pushing the deployment to an IoT Edge device.

Select your device

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. Select **Devices** under the **Device management** menu.
3. Select the target IoT Edge device from the list.
4. Select **Set Modules**.

Configure a deployment manifest

A deployment manifest is a JSON document that describes which modules to deploy, how data flows between the modules, and desired properties of the module twins. The Azure portal has a wizard that walks you through creating a deployment manifest. It has three steps organized into tabs: **Modules**, **Routes**, and **Review + Create**.

Add modules

1. In the **IoT Edge Modules** section of the page, select the **Add** dropdown and select **IoT Edge Module** to display the **Add IoT Edge Module** page.
2. On the **Settings** tab, provide a name for the module and then specify the container image URI:
 - **IoT Edge Module Name:** `azureblobstorageoniotedge`
 - **Image URI:** `mcr.microsoft.com/azure-blob-storage:latest`

Add IoT Edge Module ...

iot-hub

IoT Edge module settings. [Learn more](#)

Module name *

[Settings](#) [Environment Variables](#) [Container Create Options](#) [Module Twin Settings](#)

Image URI *

Restart Policy *

Desired Status *

Don't select **Add** until you've specified values on the **Module Settings**, **Container Create Options**, and **Module Twin Settings** tabs as described in this procedure.

ⓘ Important

Azure IoT Edge is case-sensitive when you make calls to modules, and the Storage SDK also defaults to lowercase. Changing the name to lowercase helps to ensure that your connections to the Azure Blob Storage on IoT Edge module aren't interrupted.

3. Open the **Container Create Options** tab.
4. Copy and paste the following JSON into the box, to provide storage account information and a mount for the storage on your device.

JSON

```
{  
  "Env": [  
    "LOCAL_STORAGE_ACCOUNT_NAME=<local storage account name>",  
    "LOCAL_STORAGE_ACCOUNT_KEY=<local storage account key>"  
  ],  
  "HostConfig": {  
    "Binds": [  
      "<mount>"  
    ],  
    "PortBindings": {  
      "11002/tcp": [{"HostPort": "11002"}]  
    }  
  }  
}
```

Add IoT Edge Module

iot-hub

IoT Edge module settings. [Learn more](#)

Module name *

azureblobstorageoniotedge

Settings Environment Variables **Container Create Options** Module Twin Settings

Create options direct the creation of the IoT Edge module Docker container. [View all options](#).

```
1  {
2    "Env": [
3      "LOCAL_STORAGE_ACCOUNT_NAME=<local storage account name>",
4      "LOCAL_STORAGE_ACCOUNT_KEY=<local storage account key>"
5    ],
6    "HostConfig": {
7      "Binds": [
8        "<mount>"
9      ],
10     "PortBindings": {
11       "11002/tcp": [{"HostPort": "11002"}]
12     }
13   }
```

5. Update the JSON that you copied into **Container Create Options** with the following information:

- Replace `<local storage account name>` with a name that you can remember. Account names should be 3 to 24 characters long, with lowercase letters and numbers. No spaces.
- Replace `<local storage account key>` with a 64-byte base64 key. You can generate a key with tools like [GeneratePlus](#). You use these credentials to access the blob storage from other modules.
- Replace `<mount>` according to your container operating system. Provide the name of a [volume](#) or the absolute path to an existing directory on your IoT Edge device where the blob module stores its data. The storage mount maps a location on your device that you provide to a set location in the module.

For Linux containers, the format is `<your storage path or volume>:/blobroot`. For example:

- Use [volume mount](#): `my-volume:/blobroot`

- Use [bind mount](#): /srv/containerdata:/blobroot. Make sure to follow the steps to grant directory access to the container user

 **Important**

- Do not change the second half of the storage mount value, which points to a specific location in the Blob Storage on IoT Edge module. The storage mount must always end with **:/blobroot** for Linux containers.
- IoT Edge does not remove volumes attached to module containers. This behavior is by design, as it allows persisting the data across container instances such as upgrade scenarios. However, if these volumes are left unused, then it may lead to disk space exhaustion and subsequent system errors. If you use docker volumes in your scenario, then we encourage you to use docker tools such as [docker volume prune](#) and [docker volume rm](#) to remove the unused volumes, especially for production scenarios.

6. On the **Module Twin Settings** tab, copy the following JSON and paste it into the box.

JSON

```
{
  "deviceAutoDeleteProperties": {
    "deleteOn": <true, false>,
    "deleteAfterMinutes": <timeToLiveInMinutes>,
    "retainWhileUploading": <true, false>
  },
  "deviceToCloudUploadProperties": {
    "uploadOn": <true, false>,
    "uploadOrder": "<NewestFirst, OldestFirst>",
    "cloudStorageConnectionString": "DefaultEndpointsProtocol=https;AccountName=<your Azure Storage Account Name>;AccountKey=<your Azure Storage Account Key>; EndpointSuffix=<your end point suffix>",
    "storageContainersForUpload": {
      "<source container name1>": {
        "target": "<your-target-container-name>"
      }
    },
    "deleteAfterUpload": <true, false>
  }
}
```

7. Configure each property with an appropriate value, as indicated by the placeholders. If you're using the IoT Edge simulator, set the values to the related environment variables for these properties as described by `deviceToCloudUploadProperties` and `deviceAutoDeleteProperties`.

 **Tip**

The name for your `target` container has naming restrictions, for example using a `$` prefix is unsupported. To see all restrictions, view [Container Names](#).

 **Note**

If your container target is unnamed or null within `storageContainersForUpload`, a default name will be assigned to the target. If you wanted to stop uploading to a container, it must be removed completely from `storageContainersForUpload`. For more information, see the `deviceToCloudUploadProperties` section of [Store data at the edge with Azure Blob Storage on IoT Edge](#).

Add IoT Edge Module

iot-hub

IoT Edge module settings. [Learn more](#)

Module name *

azureblobstorageoniotedge

Settings

Environment Variables

Container Create Options

Module Twin Settings

Module twin desired properties will be updated to reflect listed settings.

```
1  {
2      "deviceAutoDeleteProperties": {
3          "deleteOn": false,
4          "deleteAfterMinutes": 240,
5          "retainWhileUploading": true
6      },
7      "deviceToCloudUploadProperties": {
8          "uploadOn": true,
9          "uploadOrder": "OldestFirst",
10         "cloudStorageConnectionString": "DefaultEndpointsProtocol=https;AccountName=yourazuredgestorage;AccountKey=yourkey;ContainerName=yourcontainer",
11         "storageContainersForUpload": {
12             "yourazuredgestorage": {
13                 "target": "yourcontainer"
14             }
15         },
16         "deleteAfterUpload": false
}
```

For information on configuring deviceToCloudUploadProperties and deviceAutoDeleteProperties after your module is deployed, see [Edit the Module Twin](#). For more information about desired properties, see [Define or update desired properties](#).

8. Select Add.

9. Select **Next: Routes** to continue to the routes section.

Specify routes

Keep the default routes and select **Next: Review + create** to continue to the review section.

Review deployment

The review section shows you the JSON deployment manifest that was created based on your selections in the previous two sections. There are also two modules declared that

you didn't add: `$edgeAgent` and `$edgeHub`. These two modules make up the [IoT Edge runtime](#) and are required defaults in every deployment.

Review your deployment information, then select **Create**.

Verify your deployment

After you create the deployment, you return to the **Devices** page of your IoT hub.

1. Select the IoT Edge device that you targeted with the deployment to open its details.
2. In the device details, verify that the blob storage module is listed as both **Specified in deployment** and **Reported by device**.

It might take a few moments for the module to be started on the device and then reported back to IoT Hub. Refresh the page to see an updated status.

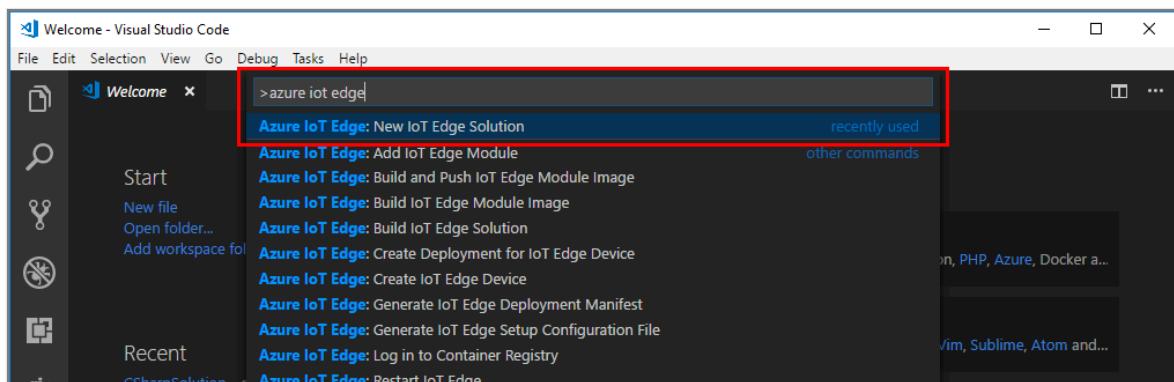
Deploy from Visual Studio Code

Azure IoT Edge provides templates in Visual Studio Code to help you develop edge solutions. Use the following steps to create a new IoT Edge solution with a blob storage module and to configure the deployment manifest.

ⓘ Important

The Azure IoT Edge Visual Studio Code extension is in [maintenance mode](#).

1. Select **View > Command Palette**.
2. In the command palette, enter and run the command **Azure IoT Edge: New IoT Edge solution**.



Follow the prompts in the command palette to create your solution.

Field	Value
Select folder	Choose the location on your development machine for Visual Studio Code to create the solution files.
Provide a solution name	Enter a descriptive name for your solution or accept the default EdgeSolution .
Select module template	Choose Existing Module (Enter full image URL) .
Provide a module name	<p>Enter an all-lowercase name for your module, like azureblobstorageoniotedge.</p> <p>It's important to use a lowercase name for the Azure Blob Storage on IoT Edge module. IoT Edge is case-sensitive when referring to modules, and the Storage SDK defaults to lowercase.</p>
Provide Docker image for the module	Provide the image URI: mcr.microsoft.com/azure-blob-storage:latest

Visual Studio Code takes the information you provided, creates an IoT Edge solution, and then loads it in a new window. The solution template creates a deployment manifest template that includes your blob storage module image, but you need to configure the module's create options.

3. Open *deployment.template.json* in your new solution workspace and find the **modules** section. Make the following configuration changes:

- a. Copy and paste the following code into the `createOptions` field for the blob storage module:

JSON

```

```json
"Env": [
 "LOCAL_STORAGE_ACCOUNT_NAME=<local storage account name>",
 "LOCAL_STORAGE_ACCOUNT_KEY=<local storage account key>"
],
"HostConfig": {
 "Binds": ["<mount>"],
 "PortBindings": {
 "11002/tcp": [{"HostPort": "11002"}]
 }
}
```

```

```
"modules": [],
  "azureblobstorageoniotedge": {
    "version": "1.0",
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
      "image": "mcr.microsoft.com/azure-blob-storage:latest",
      "createOptions": {
        "Env": [
          "LOCAL_STORAGE_ACCOUNT_NAME=<local storage account name>",
          "LOCAL_STORAGE_ACCOUNT_KEY=<local storage account key>"
        ],
        "HostConfig": {
          "Binds": [
            "<mount>"
          ],
          "PortBindings": {
            "11002/tcp": [
              {
                "HostPort": "11002"
              }
            ]
          }
        }
      }
    }
  }
}
```

4. Replace `<local storage account name>` with a name that you can remember.

Account names should be 3 to 24 characters long, with lowercase letters and numbers. No spaces.

5. Replace `<local storage account key>` with a 64-byte base64 key. You can generate a key with tools like [GeneratePlus](#). You use these credentials to access the blob storage from other modules.

6. Replace `<mount>` according to your container operating system. Provide the name of a [volume](#) or the absolute path to a directory on your IoT Edge device where you want the blob module to store its data. The storage mount maps a location on your device that you provide to a set location in the module.

For Linux containers, the format is `<your storage path or volume>:/blobroot`. For example:

- Use [volume mount](#): `my-volume:/blobroot`
- Use [bind mount](#): `/srv/containerdata:/blobroot`. Make sure to follow the steps to grant directory access to the container user

ⓘ Important

- Do not change the second half of the storage mount value, which points to a specific location in the Blob Storage on IoT Edge module. The storage mount must always end with `:/blobroot` for Linux containers.
- IoT Edge does not remove volumes attached to module containers. This behavior is by design, as it allows persisting the data across container instances such as upgrade scenarios. However, if these volumes are left unused, then it may lead to disk space exhaustion and subsequent system errors. If you use docker volumes in your scenario, then we encourage you to use docker tools such as [docker volume prune](#) and [docker volume rm](#) to remove the unused volumes, especially for production scenarios.

7. Configure `deviceToCloudUploadProperties` and `deviceAutoDeleteProperties` for your module by adding the following JSON to the `deployment.template.json` file. Configure each property with an appropriate value and save the file. If you're using the IoT Edge simulator, set the values to the related environment variables for these properties, which you can find in the explanation section of `deviceToCloudUploadProperties` and `deviceAutoDeleteProperties`

JSON

```
"<your azureblobstorageoniotedge module name>":{  
    "properties.desired": {  
        "deviceAutoDeleteProperties": {  
            "deleteOn": <true, false>,  
            "deleteAfterMinutes": <timeToLiveInMinutes>,  
            "retainWhileUploading": <true, false>  
        },  
        "deviceToCloudUploadProperties": {  
            "uploadOn": <true, false>,  
            "uploadOrder": "<NewestFirst, OldestFirst>",  
            "cloudStorageConnectionString":  
                "DefaultEndpointsProtocol=https;AccountName=<your Azure Storage Account Name>;AccountKey=<your Azure Storage Account Key>;EndpointSuffix=<your end point suffix>",  
            "storageContainersForUpload": {  
                "<source container name1>": {  
                    "target": "<target container name1>"  
                }  
            },  
            "deleteAfterUpload": <true, false>  
        }  
    }  
}
```

```
}
```

```
"azureblobstorageoniotedge": [
  "properties.desired": {
    "deviceToCloudUploadProperties": {
      "uploadOn": true,
      "uploadOrder": "OldestFirst",
      "cloudStorageConnectionString": "<AzureConnectionString>",
      "storageContainersForUpload": {
        "cont1": {
          "target": "edge-cont1"
        },
        "cont2": {
          "target": "%m-%c"
        },
        "cont3": {}
      },
      "deleteAfterUpload": true
    },
    "deviceAutoDeleteProperties": {
      "deleteOn": true,
      "deleteAfterMinutes": 5,
      "retainWhileUploading": true
    }
  }
]
```

For information on configuring deviceToCloudUploadProperties and deviceAutoDeleteProperties after your module is deployed, see [Edit the Module Twin](#). For more information about container create options, restart policy, and desired status, see [EdgeAgent desired properties](#).

8. Save the *deployment.template.json* file.
9. Right-click **deployment.template.json** and select **Generate IoT Edge deployment manifest**.
10. Visual Studio Code takes the information that you provided in *deployment.template.json* and uses it to create a new deployment manifest file. The deployment manifest is created in a new **config** folder in your solution workspace. Once you have that file, you can follow the steps in [Deploy Azure IoT Edge modules with Azure CLI 2.0](#).

Deploy multiple module instances

If you want to deploy multiple instances of the Azure Blob Storage on IoT Edge module, you need to provide a different storage path and change the `HostPort` value that the module binds to. The blob storage modules always expose port 11002 in the container, but you can declare which port it's bound to on the host.

Edit **Container Create Options** (in the Azure portal) or the `createOptions` field (in the `deployment.template.json` file in Visual Studio Code) to change the `HostPort` value:

JSON

```
"PortBindings":{  
    "11002/tcp": [{"HostPort": "<port number>"}]  
}
```

When you connect to additional blob storage modules, change the endpoint to point to the updated host port.

Configure proxy support

If your organization is using a proxy server, you need to configure proxy support for the `edgeAgent` and `edgeHub` runtime modules. This process involves two tasks:

- Configure the runtime daemons and the IoT Edge agent on the device.
- Set the `HTTPS_PROXY` environment variable for modules in the deployment manifest JSON file.

This process is described in [Configure an IoT Edge device to communicate through a proxy server](#).

In addition, a blob storage module also requires the `HTTPS_PROXY` setting in the manifest deployment file. You can directly edit the deployment manifest file, or use the Azure portal.

1. Navigate to your IoT Hub in the Azure portal and select **Devices** under the **Device management** menu
2. Select the device with the module to configure.
3. Select **Set Modules**.
4. In the **IoT Edge Modules** section of the page, select the blob storage module.
5. On the **Update IoT Edge Module** page, select the **Environment Variables** tab.
6. Add `HTTPS_PROXY` for the **Name** and your proxy URL for the **Value**.

Update IoT Edge Module

Specify the settings for an IoT Edge custom module.
[Learn more](#)

IoT Edge Module Name *

[Module Settings](#) **Environment Variables** [Container Create Options](#) [Module Twin Settings](#)

Environment variables provide supplemental information to a module facilitating the configuration process.

| NAME | VALUE |
|---------------|------------------------|
| HTTPS_PROXY | https://proxy.com:3128 |
| Variable name | Variable value |

[Update](#)

[Cancel](#)

7. Select **Update**, then **Review + Create**.
8. See the proxy is added to the module in deployment manifest and select **Create**.
9. Verify the setting by selecting the module from the device details page, and on the lower part of the **IoT Edge Modules Details** page select the **Environment Variables** tab.

| IoT Edge Module Settings | Container Create Options | Environment Variables |
|--------------------------|--------------------------|---------------------------------------|
| Setting Name | Desired Value | Reported Value |
| HTTPS_PROXY | https://proxy.com:3128 | not defined |

Next steps

Learn more about [Azure Blob Storage on IoT Edge](#).

For more information about how deployment manifests work and how to create them, see [Understand how IoT Edge modules can be used, configured, and reused](#).

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback  | Get help at Microsoft Q&A

What is Azure SQL Edge?

Article • 09/21/2024

ⓘ Important

Azure SQL Edge will be retired on September 30, 2025. For more information and migration options, see the [Retirement notice](#).

ⓘ Note

Azure SQL Edge no longer supports the ARM64 platform.

Azure SQL Edge is an optimized relational database engine geared for IoT and IoT Edge deployments. It provides capabilities to create a high-performance data storage and processing layer for IoT applications and solutions. Azure SQL Edge provides capabilities to stream, process, and analyze relational and nonrelational data such as JSON, graph and time-series data, which makes it the right choice for various modern IoT applications.

Azure SQL Edge is built on the latest versions of the [SQL Server Database Engine](#), which provides industry-leading performance, security and query processing capabilities. Since Azure SQL Edge is built on the same engine as [SQL Server](#) and [Azure SQL](#), it provides the same Transact-SQL (T-SQL) programming surface area that makes development of applications or solutions easier and faster, and makes application portability between IoT Edge devices, data centers and the cloud straight forward.

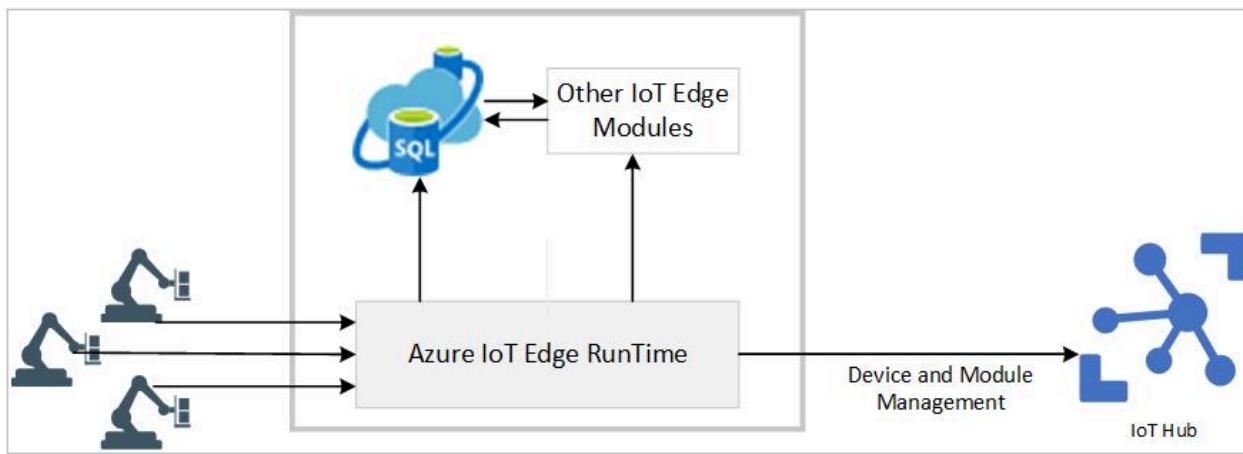
What is Azure SQL Edge video on Channel 9:

<https://learn.microsoft.com/shows/Data-Exposed/What-is-Azure-SQL-Edge/player>

Deployment models

Azure SQL Edge supports two deployment modes.

- Connected deployment through Azure IoT Edge: Azure SQL Edge is available as a module for [Azure IoT Edge](#). For more information, see [Deploy Azure SQL Edge](#).



- Disconnected deployment: Azure SQL Edge container images can be pulled from Docker hub and deployed either as a standalone container or on a Kubernetes cluster. For more information, see [Deploy Azure SQL Edge with Docker](#) and [Deploy an Azure SQL Edge container in Kubernetes](#).

Editions of SQL Edge

SQL Edge is available with two different editions or software plans. These editions have identical feature sets and only differ in terms of their usage rights and the amount of CPU/memory they support.

[] [Expand table](#)

| Plan | Description |
|--------------------------|--|
| Azure SQL Edge Developer | Development only SKU. Each SQL Edge container is limited to a maximum of 4 CPU cores and 32 GB of memory |
| Azure SQL Edge | Production SKU. Each SQL Edge container is limited to a maximum of 8 CPU cores, and 64 GB of memory. |

Price and availability

Azure SQL Edge is generally available. For more information on the pricing and availability in specific regions, see [Azure SQL Edge](#).

i Important

To understand the feature differences between Azure SQL Edge and SQL Server, as well as the differences among different Azure SQL Edge options, see [Supported features of Azure SQL Edge](#).

Streaming capabilities

Azure SQL Edge provides built in streaming capabilities for real-time analytics and complex event-processing. The streaming capability is built using the same constructs as [Azure Stream Analytics](#) and similar capabilities as [Azure Stream Analytics on IoT Edge](#).

The streaming engine for Azure SQL Edge is designed for low-latency, resiliency, efficient use of bandwidth and compliance.

For more information on data streaming in SQL Edge, see [Data Streaming](#).

Machine learning and artificial intelligence capabilities

Azure SQL Edge provides built-in machine learning and analytics capabilities by integrating the open format ONNX (Open Neural Network Exchange) runtime, which allows exchange of deep learning and neural network models between different frameworks. For more information on ONNX, see [here](#). ONNX runtime provides the flexibility to develop models in a language or tools of your choice, which can then be converted to the ONNX format for execution within SQL Edge. For more information, see [Machine Learning and Artificial Intelligence with ONNX in SQL Edge](#).

Work with Azure SQL Edge

Azure SQL Edge makes developing and maintaining applications easier and more productive. Users can use all the familiar tools and skills to build great apps and solutions for their IoT Edge needs. You can develop in SQL Edge using the following tools:

- [The Azure portal](#) - A web-based application for managing all Azure services.
- [SQL Server Management Studio](#) - A free, downloadable client application for managing any SQL infrastructure, from SQL Server to SQL Database.
- [SQL Server Data Tools in Visual Studio](#) - A free, downloadable client application for developing SQL Server relational databases, SQL databases, Integration Services packages, Analysis Services data models, and Reporting Services reports.
- [Azure Data Studio](#) - A free, downloadable, cross platform database tool for data professionals using the Microsoft family of on-premises and cloud data platforms on Windows, macOS, and Linux.
- [Visual Studio Code](#) - A free, downloadable, open-source code editor for Windows, macOS, and Linux. It supports extensions, including the [mssql](#)

[extension](#) for querying Microsoft SQL Server, Azure SQL Database, and Azure Synapse Analytics.

Related content

- [Deploy SQL Edge through Azure portal](#)
 - [Machine Learning and Artificial Intelligence with SQL Edge](#)
 - [Building an end-to-end IoT solution with SQL Edge](#)
-

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Deploy Azure SQL Edge

Article • 09/22/2024

ⓘ Important

Azure SQL Edge will be retired on September 30, 2025. For more information and migration options, see the [Retirement notice](#).

ⓘ Note

Azure SQL Edge no longer supports the ARM64 platform.

Azure SQL Edge is a relational database engine optimized for IoT and Azure IoT Edge deployments. It provides capabilities to create a high-performance data storage and processing layer for IoT applications and solutions. This quickstart shows you how to get started with creating an Azure SQL Edge module through Azure IoT Edge using the Azure portal.

Before you begin

- If you don't have an Azure subscription, create a [free account](#).
- Sign in to the [Azure portal](#).
- Create an [Azure IoT Hub](#).
- Create an [Azure IoT Edge device](#).

ⓘ Note

To deploy an Azure Linux VM as an IoT Edge device, see this [quickstart guide](#).

Deploy Azure SQL Edge Module using IoT Hub

Azure SQL Edge can be deployed using instructions from [Deploy modules from Azure portal](#). The image URI for Azure SQL Edge is `mcr.microsoft.com/azure-sql-edge:latest`.

1. On the [Add IoT Edge Module](#) page, specify the desired values for the *IoT Edge Module Name*, *Image URI*, *Restart Policy* and *Desired Status*.

Use the following image URI depending on the edition you want to deploy:

- *Developer edition* - mcr.microsoft.com/azure-sql-edge/developer
- *Premium edition* - mcr.microsoft.com/azure-sql-edge/premium

2. On the *Environment Variables* section of the **Add IoT Edge Module** page, specify the desired values for the environment variables. For a complete list of Azure SQL Edge environment variables, see [Configure using environment variables](#).

[Expand table](#)

| Parameter | Description |
|-------------------|---|
| ACCEPT_EULA | Set this value to <code>Y</code> to accept the End-User Licensing Agreement |
| MSSQL_SA_PASSWORD | Set the value to specify a strong password for the SQL Edge admin account. |
| MSSQL_LCID | Set the value to set the desired language ID to use for SQL Edge. For example, 1036 is French. |
| MSSQL_COLLATION | Set the value to set the default collation for SQL Edge. This setting overrides the default mapping of language ID (LCID) to collation. |

3. On the *Container Create Options* section of the **Add IoT Edge Module** page, set the options as per requirement.

- **Host Port**

Map the specified host port to port 1433 (default SQL port) in the container.

- **Binds and Mounts**

If you need to deploy more than one SQL Edge module, ensure that you update the mounts option to create a new source and target pair for the persistent volume. For more information on mounts and volume, see [Use volumes](#) on Docker documentation.

JSON

```
{
  "HostConfig": {
    "CapAdd": [
      "SYS_PTRACE"
    ],
    "Binds": [
      "sqlvolume:/sqlvolume"
    ],
    "PortBindings": {
      "1433/tcp": [
        {
          "HostPort": 1433
        }
      ]
    }
  }
}
```

```
        "1433/tcp": [
            {
                "HostPort": "1433"
            }
        ],
        "Mounts": [
            {
                "Type": "volume",
                "Source": "sqlvolume",
                "Target": "/var/opt/mssql"
            }
        ]
    },
    "Env": [
        "MSSQL_AGENT_ENABLED=TRUE",
        "ClientTransportType=AMQP_TCP_Only",
        "PlanId=asde-developer-on-iot-edge"
    ]
}
```

ⓘ Important

Set the `PlanId` environment variable based on the edition installed.

- *Developer edition* - `asde-developer-on-iot-edge`
- *Premium edition* - `asde-premium-on-iot-edge`

If this value is set incorrectly, the Azure SQL Edge container fails to start.

⚠ Warning

If you reinstall the module, remember to remove any existing bindings first, otherwise your environment variables will not be updated.

4. On the **Add IoT Edge Module** page, select **Add**.
5. On the **Set modules on device** page, select **Next: Routes** > if you need to define routes for your deployment. Otherwise select **Review + Create**. For more information on configuring routes, see [Deploy modules and establish routes in IoT Edge](#).
6. On the **Set modules on device** page, select **Create**.

Connect to Azure SQL Edge

The following steps use the Azure SQL Edge command-line tool, **sqlcmd**, inside the container to connect to Azure SQL Edge.

ⓘ Note

SQL Server command line tools, including **sqlcmd**, aren't available inside the ARM64 version of Azure SQL Edge containers.

1. Use the `docker exec -it` command to start an interactive bash shell inside your running container. In the following example, `AzureSQLEdge` is name specified by the `Name` parameter of your IoT Edge Module.

```
Bash
```

```
sudo docker exec -it AzureSQLEdge "bash"
```

2. Once inside the container, connect locally with the **sqlcmd** tool. **sqlcmd** isn't in the path by default, so you have to specify the full path.

```
Bash
```

```
/opt/mssql-tools/bin/sqlcmd -S localhost -U SA -P "  
<YourNewStrong@Passw0rd>"
```

💡 Tip

You can omit the password on the command-line to be prompted to enter it.

3. If successful, you should get to a **sqlcmd** command prompt: `1>`.

Create and query data

The following sections walk you through using **sqlcmd** and Transact-SQL to create a new database, add data, and run a query.

Create a new database

The following steps create a new database named `TestDB`.

1. From the **sqlcmd** command prompt, paste the following Transact-SQL command to create a test database:

```
SQL  
  
CREATE DATABASE TestDB;  
GO
```

2. On the next line, write a query to return the name of all of the databases on your server:

```
SQL  
  
SELECT name from sys.databases;  
GO
```

Insert data

Next, create a new table called `Inventory`, and insert two new rows.

1. From the **sqlcmd** command prompt, switch context to the new `TestDB` database:

```
SQL  
  
USE TestDB;
```

2. Create new table named `Inventory`:

```
SQL  
  
CREATE TABLE Inventory (id INT, name NVARCHAR(50), quantity INT)
```

3. Insert data into the new table:

```
SQL  
  
INSERT INTO Inventory  
VALUES (1, 'banana', 150);  
  
INSERT INTO Inventory  
VALUES (2, 'orange', 154);
```

4. Type `go` to execute the previous commands:

```
SQL
```

```
GO
```

Select data

Now, run a query to return data from the `Inventory` table.

1. From the `sqlcmd` command prompt, enter a query that returns rows from the `Inventory` table where the quantity is greater than 152:

```
SQL
```

```
SELECT * FROM Inventory WHERE quantity > 152;
```

2. Execute the command:

```
SQL
```

```
GO
```

Exit the `sqlcmd` command prompt

1. To end your `sqlcmd` session, type `QUIT`:

```
SQL
```

```
QUIT
```

2. To exit the interactive command-prompt in your container, type `exit`. Your container continues to run after you exit the interactive bash shell.

Connect from outside the container

You can connect and run SQL queries against your Azure SQL Edge instance from any external Linux, Windows, or macOS tool that supports SQL connections. For more information on connecting to a SQL Edge container from outside, see [Connect and Query Azure SQL Edge](#).

In this quickstart, you deployed a SQL Edge Module on an IoT Edge device.

Related content

- Machine Learning and Artificial Intelligence with ONNX in SQL Edge
 - Building an end to end IoT Solution with SQL Edge using IoT Edge
 - Data Streaming in Azure SQL Edge
 - Troubleshoot deployment errors
-

Feedback

Was this page helpful?



Provide product feedback [↗](#) | Get help at Microsoft Q&A

Continuous integration and continuous deployment to Azure IoT Edge devices

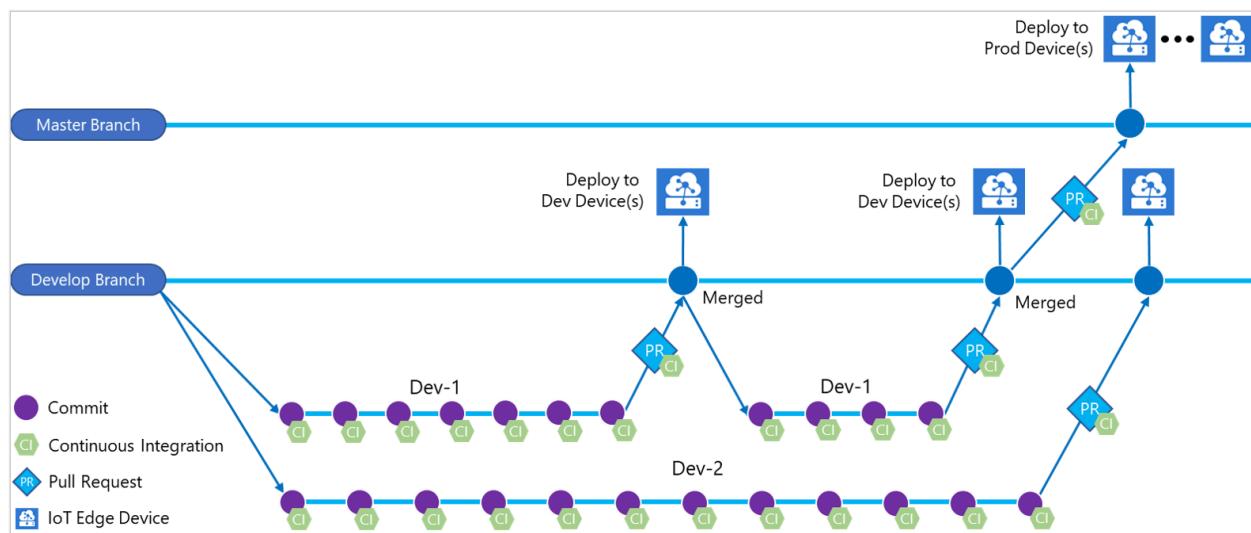
Article • 04/09/2024

Applies to: IoT Edge 1.5 IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

You can easily adopt DevOps with your Azure IoT Edge applications with the built-in Azure IoT Edge tasks in Azure Pipelines. This article demonstrates how you can use Azure Pipelines to build, test, and deploy Azure IoT Edge modules using YAML. Alternatively, you can [use the classic editor](#).



In this article, you learn how to use the built-in [Azure IoT Edge tasks](#) for Azure Pipelines to create build and release pipelines for your IoT Edge solution. Each Azure IoT Edge task added to your pipeline implements one of the following four actions:

[\[\]](#) [Expand table](#)

| Action | Description |
|---------------------|--|
| Build module images | Takes your IoT Edge solution code and builds the container images. |
| Push module images | Pushes module images to the container registry you specified. |

| Action | Description |
|------------------------------|--|
| Generate deployment manifest | Takes a deployment.template.json file and the variables, then generates the final IoT Edge deployment manifest file. |
| Deploy to IoT Edge devices | Creates IoT Edge deployments to one or more IoT Edge devices. |

Unless otherwise specified, the procedures in this article do not explore all the functionality available through task parameters. For more information, see the following resources:

- [Task version](#)
- [Control Options](#)
- [Environment Variables](#)
- [Output variables](#)

Prerequisites

- An Azure Repos repository. If you don't have one, you can [Create a new Git repo in your project](#). For this article, we created a repository called **IoTEdgeRepo**.
- An IoT Edge solution committed and pushed to your repository. If you want to create a new sample solution for testing this article, follow the steps in [Develop Azure IoT Edge modules using Visual Studio Code](#). For this article, we created a solution in our repository called **IoTEdgeSolution**, which has the code for a module named **filtermodule**.

For this article, all you need is the solution folder created by the IoT Edge templates in either Visual Studio Code or Visual Studio. You don't need to build, push, deploy, or debug this code before proceeding. You'll set up those processes in Azure Pipelines.

Know the path to the **deployment.template.json** file in your solution, which is used in several steps. If you're unfamiliar with the role of the deployment template, see [Learn how to deploy modules and establish routes](#).

Tip

If you're creating a new solution, clone your repository locally first. Then, when you create the solution you can choose to create it directly in the repository folder. You can easily commit and push the new files from there.

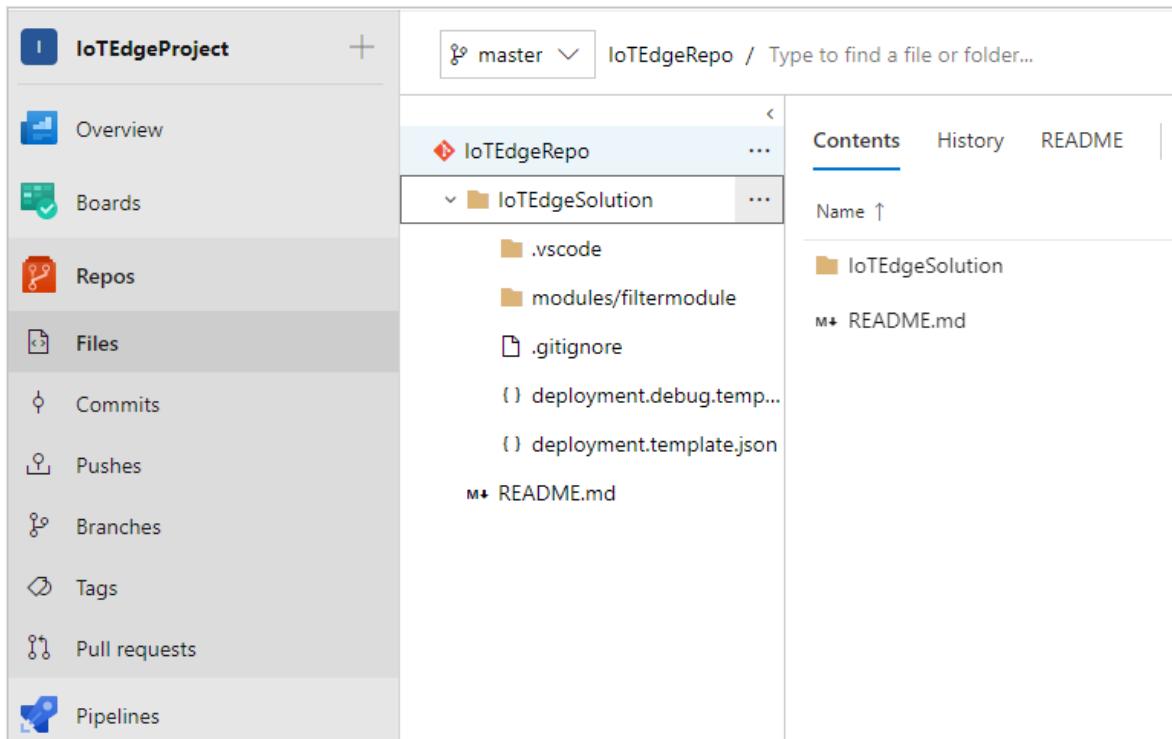
- A container registry where you can push module images. You can use [Azure Container Registry](#) or a third-party registry.
- An active Azure [IoT hub](#) with at least two IoT Edge devices for testing the separate test and production deployment stages. You can follow the quickstart articles to create an IoT Edge device on [Linux](#) or [Windows](#)

For more information about using Azure Repos, see [Share your code with Visual Studio and Azure Repos](#).

Create a build pipeline for continuous integration

In this section, you create a new build pipeline. You configure the pipeline to run automatically when you check in any changes to the sample IoT Edge solution and to publish build logs.

1. Sign in to your Azure DevOps organization (<https://dev.azure.com/{your organization}>) and open the project that contains your IoT Edge solution repository.



2. From the left pane menu in your project, select **Pipelines**. Select **Create Pipeline** at the center of the page. Or, if you already have build pipelines, select the **New pipeline** button in the top right.

The screenshot shows the Azure DevOps Pipelines interface. On the left, there's a sidebar with icons for Overview, Boards, Repos, Pipelines (which is selected), Environments, Releases, Library, Task groups, and Deployment groups. The main area is titled 'Pipelines' and shows a table of 'Recently run pipelines'. The table has two columns: 'Pipeline' and 'Last run'. Three pipelines are listed:

| Pipeline | Last run |
|---|---|
| iot-edge-yaml-pipeline-test #20200521.1 • Set up CI with... | Thursday
Manually triggered by azure-p...
12s |
| iot-edge-yaml-pipeline-test #20200521.1 • Updated iote... | Thursday
Individual CI for azure-pip...
9m 1s |
| iot-edge-yaml-pipeline-test #553 • Add yaml pipeline | Apr 9
Manually triggered for mas...
4m 23s |

A red box highlights the 'New pipeline' button in the top right corner.

3. On the **Where is your code?** page, select **Azure Repos Git YAML**. If you wish to use the classic editor to create your project's build pipelines, see the [classic editor guide](#).
4. Select the repository you are creating a pipeline for.

The screenshot shows the 'Select a repository' step in the pipeline creation wizard. The top navigation bar includes 'Connect', 'Select' (which is highlighted in blue), 'Configure', and 'Review'. Below the navigation, it says 'New pipeline' and 'Select a repository'. A search bar is labeled 'Filter by keywords' with the text 'iot-edge-yaml-pipeline-test'. A single repository entry, 'iot-edge-yaml-pipeline-test', is shown in a box with a red border.

5. On the **Configure your pipeline** page, select **Starter pipeline**. If you have a preexisting Azure Pipelines YAML file you wish to use to create this pipeline, you can select **Existing Azure Pipelines YAML file** and provide the branch and path in the repository to the file.

New pipeline

Configure your pipeline

- Docker Build a Docker image
- Docker Build and push an image to Azure Container Registry
- Deploy to Azure Kubernetes Service** Build and push image to Azure Container Registry; Deploy to Azure Kubernetes Service
- Deploy to Kubernetes - Review app with Azure DevSpaces Build and push image to Azure Container Registry; Deploy to Azure Kubernetes Services and setup Review App with Azure DevSpaces
- .NET Desktop Build and run tests for .NET Desktop or Windows classic desktop solutions.
- Starter pipeline** Start with a minimal pipeline that you can customize to build and deploy your code.
- Existing Azure Pipelines YAML file** Select an Azure Pipelines YAML file in any branch of the repository.

Show more

- On the **Review your pipeline YAML** page, you can select the default name `azure-pipelines.yml` to rename your pipeline's configuration file.

Select **Show assistant** to open the **Tasks** palette.

Pipelines

✓ Connect ✓ Select ✓ Configure Review

New pipeline

Review your pipeline YAML

Variables Save and run

Show assistant

```

1 # Starter pipeline
2 # Start with a minimal pipeline that you can customize to build and deploy your code.
3 # Add steps that build, run tests, deploy, and more:
4 # https://aka.ms/yaml
5
6 trigger:
7 - master
8
9 pool:
10 - vmImage: 'ubuntu-latest'
11
12 steps:
13 - script: echo Hello, world!
14 - displayName: 'Run a one-line script'
15
16 - script: |
17 | echo Add other tasks to build, test, and deploy your project.

```

- To add a task, place your cursor at the end of the YAML or wherever you want the instructions for your task to be added. Search for and select **Azure IoT Edge**. Fill out the task's parameters as follows. Then, select **Add**.

Expand table

| Parameter | Description |
|---------------------|--|
| Action | Select Build module images . |
| .template.json file | Provide the path to the deployment.template.json file in the repository that contains your IoT Edge solution. |
| Default platform | Select the appropriate operating system for your modules based on your targeted IoT Edge device. |

For more information about this task and its parameters, see [Azure IoT Edge task](#).

The screenshot shows the Azure DevOps Pipelines interface. On the left, there's a sidebar with icons for Overview, Boards, Repos, Pipelines (which is selected), Environments, Releases, Library, Task groups, Deployment groups, Test Plans, and Project settings. The main area is titled "Review your pipeline YAML" and shows a YAML script for building IoT Edge modules. To the right of the YAML, there are configuration fields for "Action" (set to "Build module images"), ".template.json file" (set to "deployment.template.json"), and "Default platform" (set to "amd64"). At the bottom right of the configuration area, there's a blue "Add" button with a red box around it.

Tip

After each task is added, the editor will automatically highlight the added lines. To prevent accidental overwriting, deselect the lines and provide a new space for your next task before adding additional tasks.

8. Repeat this process to add three more tasks with the following parameters:

- Task: **Azure IoT Edge**

[\[+\] Expand table](#)

| Parameter | Description |
|--------------------|---|
| Action | Select Push module images . |
| Container registry | Use the default type: Azure Container Registry . |

| Parameter | Description |
|--------------------------|--|
| type | |
| Azure subscription | Select your subscription. |
| Azure Container Registry | Choose the registry that you want to use for the pipeline. |
| .template.json file | Provide the path to the deployment.template.json file in the repository that contains your IoT Edge solution. |
| Default platform | Select the appropriate operating system for your modules based on your targeted IoT Edge device. |

For more information about this task and its parameters, see [Azure IoT Edge task](#).

- Task: **Copy Files**

[] [Expand table](#)

| Parameter | Description |
|---------------|---|
| Source Folder | The source folder to copy from. Empty is the root of the repo. Use variables if files are not in the repo. Example: <code>\$(agent.builddirectory)</code> . |
| Contents | Add two lines: <code>deployment.template.json</code> and <code>**/module.json</code> . |
| Target Folder | Specify the variable <code>\$(Build.ArtifactStagingDirectory)</code> . See Build variables to learn about the description. |

For more information about this task and its parameters, see [Copy files task](#).

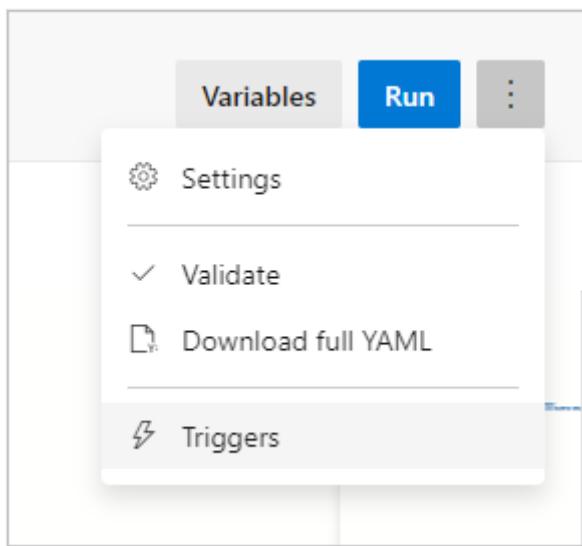
- Task: **Publish Build Artifacts**

[] [Expand table](#)

| Parameter | Description |
|---------------------------|--|
| Path to publish | Specify the variable <code>\$(Build.ArtifactStagingDirectory)</code> . See Build variables to learn about the description. |
| Artifact name | Specify the default name: <code>drop</code> |
| Artifact publish location | Use the default location: <code>Azure Pipelines</code> |

For more information about this task and its parameters, see [Publish build artifacts task](#).

9. Select **Save** from the **Save and run** dropdown in the top right.
10. The trigger for continuous integration is enabled by default for your YAML pipeline. If you wish to edit these settings, select your pipeline and select **Edit** in the top right. Select **More actions** next to the **Run** button in the top right and go to **Triggers**. **Continuous integration** shows as enabled under your pipeline's name. If you wish to see the details for the trigger, check the **Override the YAML continuous integration trigger from here** box.



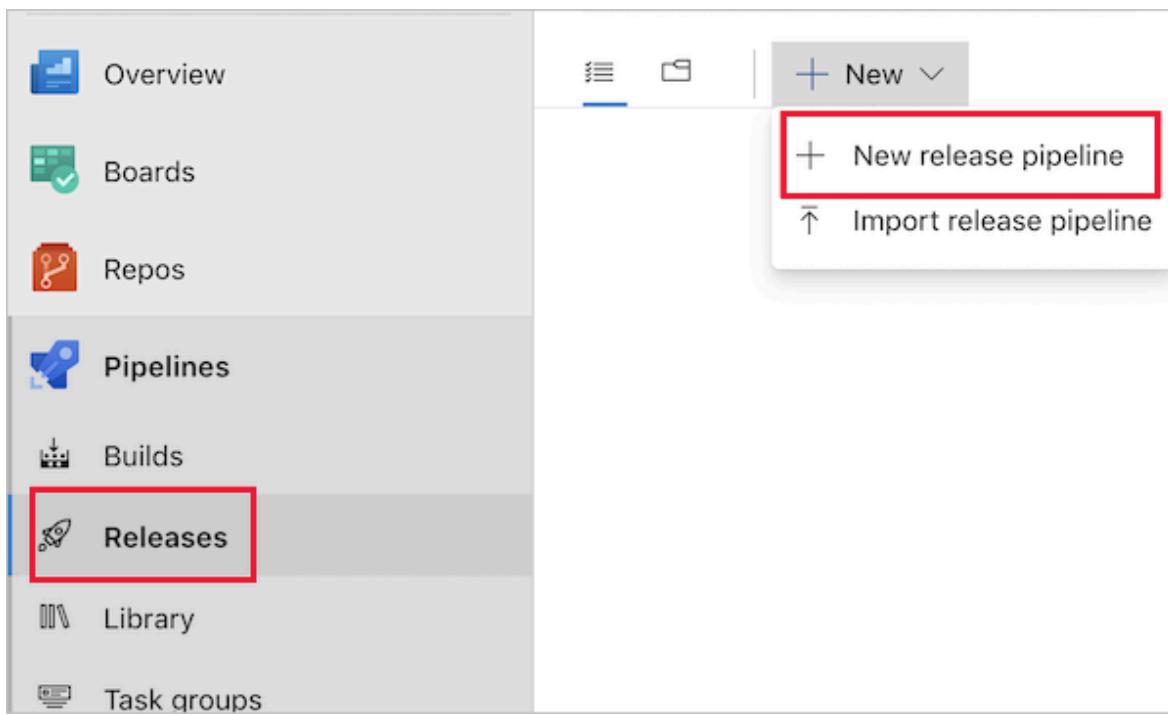
Continue to the next section to build the release pipeline.

Create a release pipeline for continuous deployment

In this section, you create a release pipeline that is configured to run automatically when your build pipeline drops artifacts, and it will show deployment logs in Azure Pipelines.

Create a new pipeline, and add a new stage:

1. In the **Releases** tab under **Pipelines**, choose **+ New pipeline**. Or, if you already have release pipelines, choose the **+ New** button and select **+ New release pipeline**.



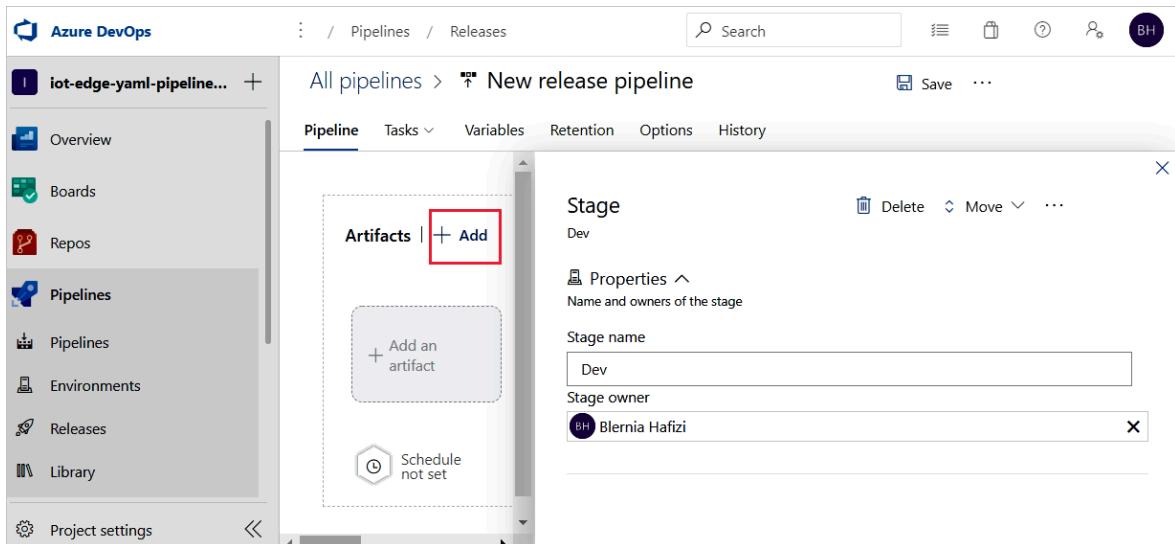
2. When prompted to select a template, choose to start with an **Empty job**.

A screenshot of the 'Select a template' dialog. It has a search bar at the top right. Below it, there's a section labeled 'Or start with an' with a button labeled 'Empty job' which is highlighted with a red box. Underneath, there's a 'Featured' section with three items: 'Azure App Service deployment' (with a globe icon), 'Deploy a Java app to Azure App Service' (with a Java icon), and 'Deploy a Node.js app to Azure App Service' (with a green hexagon icon).

3. Your new release pipeline initializes with one stage, called **Stage 1**. Rename Stage 1 to **dev** and treat it as a continuous deployment pipeline for your development environment. Usually, continuous deployment pipelines have multiple stages including **dev**, **staging**, and **prod**. You can use different names and create more based on your DevOps practice. Close the stage details window once it's renamed.

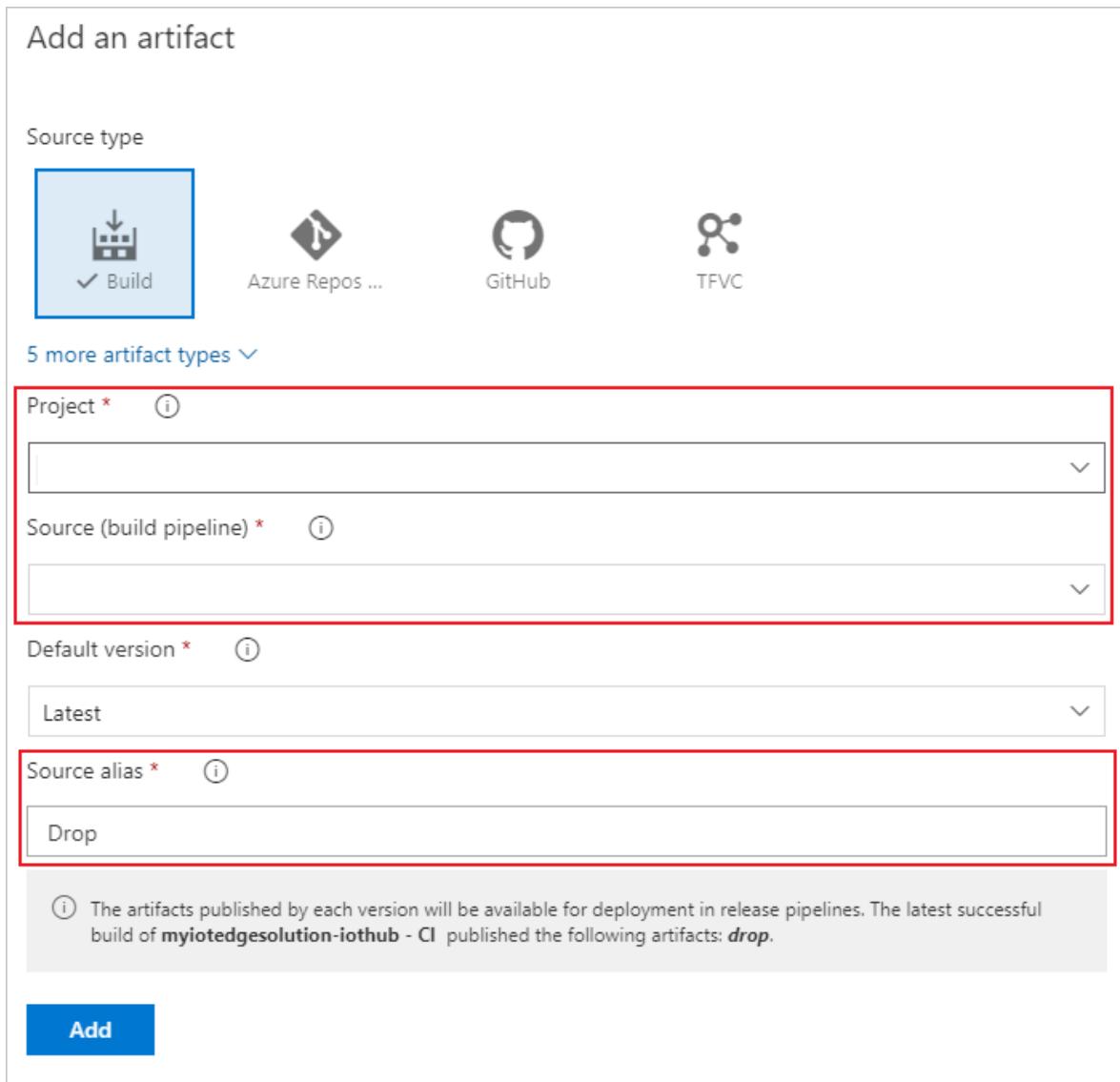
You can also rename your release pipeline by selecting the "New release pipeline" text at the top.

4. Link the release to the build artifacts that are published by the build pipeline. Click **Add** in artifacts area.



The screenshot shows the 'New release pipeline' page in Azure DevOps. On the left, there's a sidebar with 'Pipelines' selected. The main area shows a 'Stage' named 'Dev' with a 'Properties' section. In the center, under 'Artifacts', there's a '+ Add' button highlighted with a red box. Below it is a dashed box labeled '+ Add an artifact'.

5. On the **Add an artifact** page, select **Build** as the **Source type**. Choose the project and the build pipeline you created. If you wish, you can change the **Source alias** to something more descriptive. Then, select **Add**.



The screenshot shows the 'Add an artifact' form. At the top, 'Source type' is set to 'Build'. Below it, 'Project' and 'Source (build pipeline)' fields are highlighted with red boxes. Further down, 'Default version' is set to 'Latest' and 'Source alias' is set to 'Drop', both of which are also highlighted with red boxes. A note at the bottom states: '(i) The artifacts published by each version will be available for deployment in release pipelines. The latest successful build of myiotedgesolution-iohub - CI published the following artifacts: drop.' A large blue 'Add' button is at the bottom.

6. Open the artifact triggers and select the toggle to enable the continuous deployment trigger. Now, a new release will be created each time a new build is available.

The screenshot shows the 'Artifacts' section of the Azure DevOps interface. It displays two triggers: 'Drop' (selected) and 'Schedule not set'. On the right, the 'Continuous deployment trigger' is configured with the 'Enabled' toggle switch turned on (blue). A tooltip indicates it creates a release every time a new build is available. Below this, 'Build branch filters' are listed with a note that no filters have been added. The 'Pull request trigger' is shown with its 'Disabled' toggle switch turned off (black).

7. The **dev** stage is preconfigured with one job and zero tasks. From the pipeline menu, select **Tasks** then choose the **dev** stage. Select the **Agent job** and change its **Display name** to **QA**. You can configure details about the agent job, but the deployment task is platform insensitive so you can use any **Agent specification** in the chosen **Agent pool**.

The screenshot shows the 'Pipelines' screen in Azure DevOps. A new pipeline is being created under 'All pipelines > New release pipeline'. The 'Dev' stage has one 'Agent job' task. The 'QA' stage is selected, and its configuration is visible. The 'Display name' is set to 'QA', and the 'Agent pool' is set to 'Azure Pipelines'. The 'Agent Specification' is set to 'vs2017-win2016'. The 'Demands' section is empty.

8. On the QA job, select the plus sign (+) to add two tasks. Search for and add **Azure IoT Edge** twice.

9. Select the first **Azure IoT Edge** task and configure it with the following values:

[] Expand table

| Parameter | Description |
|---------------------|--|
| Display name | The display name is automatically updated when the Action field changes. |
| Action | Select Generate deployment manifest . |
| .template.json file | Specify the path:
\$(System.DefaultWorkingDirectory)/Drop/drop/deployment.template.json.
The path is published from build pipeline. |
| Default platform | Select the appropriate operating system for your modules based on your targeted IoT Edge device. |
| Output path | Put the path
\$(System.DefaultWorkingDirectory)/Drop/drop/configs/deployment.json.
This path is the final IoT Edge deployment manifest file. |

These configurations help replace the module image URLs in the `deployment.template.json` file. The **Generate deployment manifest** also helps replace the variables with the exact value you defined in the `deployment.template.json` file. In Visual Studio/Visual Studio Code, you are specifying the actual value in a `.env` file. In Azure Pipelines, you set the value in **Release Pipeline Variables** tab. Move to **Variables** tab and configure the name and value as following:

- **ACR_ADDRESS**: Your Azure Container Registry **Login server** value. You can retrieve the Login server from the Overview page of your container registry in the Azure portal.
- **ACR_PASSWORD**: Your Azure Container Registry password.
- **ACR_USER**: Your Azure Container Registry username.

If you have other variables in your project, you can specify the name and value in this tab. The **Generate deployment manifest** can only recognize the variables that are in `${VARIABLE}` flavor. Make sure you are using this flavor in your `*.template.json` files.

JSON

```
"registryCredentials": {
    "<ACR name>": { // Your Azure Container Registry **Registry name** value
        "username": "${ACR_USER}",
```

```

        "password": "${ACR_PASSWORD}",
        "address": "${ACR_ADDRESS}"
    }
}

```

The screenshot shows the Azure DevOps interface for a release pipeline. The top navigation bar indicates the path: Release > Release-4 > Stage 1. A message at the top says "Not deployed". Below this, the "Variables" tab is selected. On the left, there's a sidebar with various icons. The main area displays a table of pipeline variables:

| Name | Value |
|--------------|-----------------------------|
| ACR_ADDRESS | aziotedgeclitest.azurecr.io |
| ACR_PASSWORD | ***** |
| ACR_USER | aziotedgeclitest |

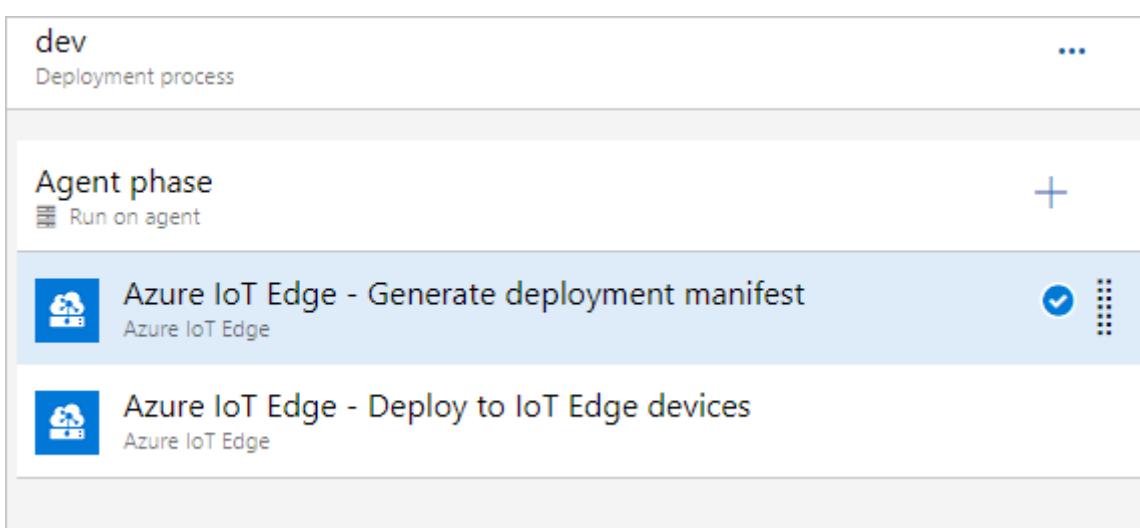
10. Select the second **Azure IoT Edge** task and configure it with the following values:

[Expand table](#)

| Parameter | Description |
|-------------------------------|---|
| Display name | The display name is automatically updated when the Action field changes. |
| Action | Select <code>Deploy to IoT Edge devices</code> . |
| Deployment file | Put the path
<code>\$(System.DefaultWorkingDirectory)/Drop/drop/configs/deployment.json</code> .
This path is the file IoT Edge deployment manifest file. |
| Azure subscription | Select the subscription that contains your IoT Hub. |
| IoT Hub name | Select your IoT hub. |
| Choose single/multiple device | Choose whether you want the release pipeline to deploy to one or multiple devices. If you deploy to a single device, enter the IoT Edge device ID . If you are deploying to multiple devices, specify the device target condition . The target condition is a filter to match a set of IoT Edge devices in IoT Hub. If you want to use device tags as the condition, you need to update your corresponding devices tags with IoT Hub device twin. Update the IoT Edge deployment ID and IoT Edge deployment priority in the advanced settings. For more information about creating a deployment for multiple devices, see Understand IoT Edge automatic deployments . |

| Parameter | Description |
|-------------------------------|---|
| Device ID or target condition | Depending on the prior selection, specify a device ID or target condition to deploy to multiple devices. |
| Advanced | For the IoT Edge deployment ID, specify <code>\$(System.TeamProject)-\$(Release.EnvironmentName)</code> . This variable maps the project and release name with your IoT Edge deployment ID. |

If your task involves using an image that resides in a private Docker Trusted Registry that isn't visible to the public cloud, you can set the **SKIP_MODULE_IMAGE_VALIDATION** environment variable to `true` to skip image validation.

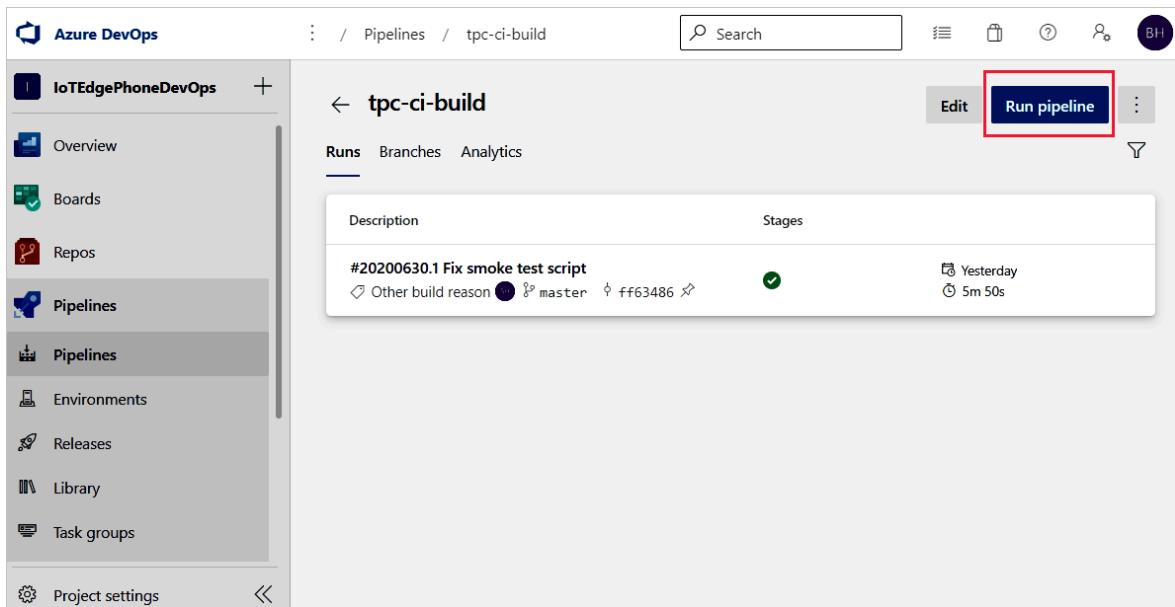


11. Select **Save** to save your changes to the new release pipeline. Return to the pipeline view by selecting **Pipeline** tab from the menu.

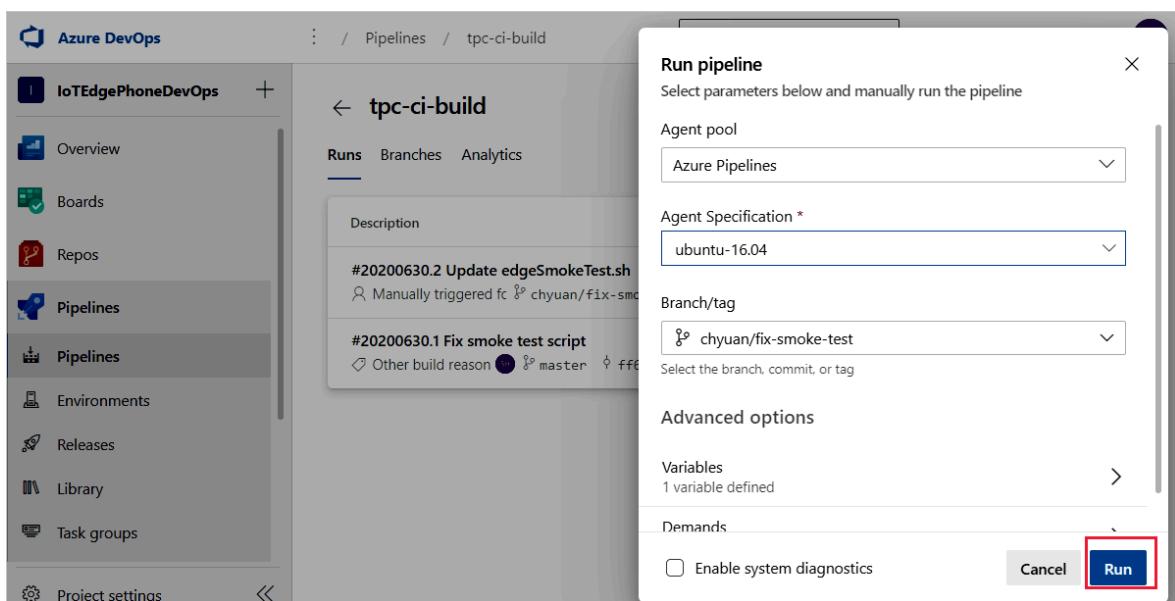
Verify IoT Edge CI/CD with the build and release pipelines

To trigger a build job, you can either push a commit to source code repository or manually trigger it. In this section, you manually trigger the CI/CD pipeline to test that it works. Then verify that the deployment succeeds.

1. From the left pane menu, select **Pipelines** and open the build pipeline that you created at the beginning of this article.
2. You can trigger a build job in your build pipeline by selecting the **Run pipeline** button in the top right.



3. Review the Run pipeline settings. Then, select Run.



4. Select Agent job 1 to watch the run's progress. You can review the logs of the job's output by selecting the job.

```

1 Starting: Azure Deployment>Create ACR
2 =====
3 Task      : Azure resource group deployment
4 Description : Deploy an Azure Resource Manager (ARM) template to a resource group and
5 Version   : 2.171.0
6 Author    : Microsoft Corporation
7 Help      : https://docs.microsoft.com/azure/devops/pipelines/tasks/deploy/azure-res
8 =====
9 Checking if the following resource group exists: IoTEdgePhoneDevOps-rg.
10 Resource group exists: true.
11 Creating deployment parameters.
12 Starting template validation.
13 Deployment name is arm-acr-20200630-162725-afee
14 Template deployment validation was completed successfully.
15 Starting Deployment.
16 Deployment name is arm-acr-20200630-162725-afee
17 Successfully deployed the template.
18 Finishing: Azure Deployment>Create ACR

```

5. If the build pipeline is completed successfully, it triggers a release to **dev** stage.

The successful **dev** release creates IoT Edge deployment to target IoT Edge devices.

| Releases | Deployments | Analytics |
|-------------------------------------|-------------------------|------------------------|
| Releases | Created | Stages |
| Release-5
20190... master | 8/14/2019, 10:40:19 PM | dev |

6. Click **dev** stage to see release logs.

| Deployment process
Succeeded | Agent phase
Succeeded | Started: 8/14/2019, 10:40:36 PM
... 3m 51s |
|---------------------------------|--|---|
| | Pool: Hosted Ubuntu 1604 · Agent: Hosted Ubuntu 1604 2
Initialize job · succeeded
Download Artifacts · succeeded
Azure IoT Edge - Generate deployment manifest · succeeded
Azure IoT Edge - Deploy to IoT Edge devices · succeeded
Finalize Job · succeeded | |

7. If your pipeline is failing, start by looking at the logs. You can view logs by navigating to the pipeline run summary and selecting the job and task. If a certain task is failing, check the logs for that task. For detailed instructions for configuring and using logs, see [Review logs to diagnose pipeline issues](#).

Next steps

- Understand the IoT Edge deployment in [Understand IoT Edge deployments for single devices or at scale](#)

- Walk through the steps to create, update, or delete a deployment in [Deploy and monitor IoT Edge modules at scale](#).

Troubleshoot your IoT Edge device

Article • 06/05/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

If you experience issues running Azure IoT Edge in your environment, use this article as a guide for troubleshooting and diagnostics.

Run the 'check' command

Your first step when troubleshooting IoT Edge should be to use the `check` command, which runs a collection of configuration and connectivity tests for common issues. The `check` command is available in [release 1.0.7](#) and later.

Note

The troubleshooting tool can't run connectivity checks if the IoT Edge device is behind a proxy server.

You can run the `check` command as follows, or include the `--help` flag to see a complete list of options:

Bash

```
sudo iotedge check
```

The troubleshooting tool runs many checks that are sorted into these three categories:

- *Configuration checks* examine details that could prevent IoT Edge devices from connecting to the cloud, including issues with the config file and the container engine.
- *Connection checks* verify that the IoT Edge runtime can access ports on the host device and that all the IoT Edge components can connect to the IoT Hub. This set

of checks returns errors if the IoT Edge device is behind a proxy.

- *Production readiness checks* look for recommended production best practices, such as the state of device certificate authority (CA) certificates and module log file configuration.

The IoT Edge check tool uses a container to run its diagnostics. The container image, `mcr.microsoft.com/azureiotedge-diagnostics:latest`, is available through the [Microsoft Container Registry](#). If you need to run a check on a device without direct access to the internet, your devices will need access to the container image.

In a scenario using nested IoT Edge devices, you can get access to the diagnostics image on downstream devices by routing the image pull through the parent devices.

Bash

```
sudo iotedge check --diagnostics-image-name <parent_device_fqdn_or_ip>:  
<port_for_api_proxy_module>/azureiotedge-diagnostics:1.2
```

For information about each of the diagnostic checks this tool runs, including what to do if you get an error or warning, see [IoT Edge troubleshoot checks](#).

Gather debug information with 'support-bundle' command

When you need to gather logs from an IoT Edge device, the most convenient way is to use the `support-bundle` command. By default, this command collects module, IoT Edge security manager and container engine logs, `iotedge check` JSON output, and other useful debug information. It compresses them into a single file for easy sharing. The `support-bundle` command is available in [release 1.0.9](#) and later.

Run the `support-bundle` command with the `--since` flag to specify how long from the past you want to get logs. For example `6h` gets logs since the last six hours, `6d` since the last six days, `6m` since the last six minutes and so on. Include the `--help` flag to see a complete list of options.

Bash

```
sudo iotedge support-bundle --since 6h
```

By default, the `support-bundle` command creates a zip file called `support_bundle.zip` in the directory where the command is called. Use the flag `--output` to specify a different

path or file name for the output.

For more information about the command, view its help information.

```
bash/cmd
```

```
iotedge support-bundle --help
```

You can also use the built-in direct method call [UploadSupportBundle](#) to upload the output of the support-bundle command to Azure Blob Storage.

 **Warning**

Output from the `support-bundle` command can contain host, device and module names, information logged by your modules etc. Please be aware of this if sharing the output in a public forum.

Review metrics collected from the runtime

The IoT Edge runtime modules produce metrics to help you monitor and understand the health of your IoT Edge devices. Add the `metrics-collector` module to your deployments to handle collecting these metrics and sending them to the cloud for easier monitoring.

For more information, see [Collect and transport metrics](#).

Check your IoT Edge version

If you're running an older version of IoT Edge, then upgrading may resolve your issue. The `iotedge check` tool checks that the IoT Edge security daemon is the latest version, but doesn't check the versions of the IoT Edge hub and agent modules. To check the version of the runtime modules on your device, use the commands `iotedge logs edgeAgent` and `iotedge logs edgeHub`. The version number is declared in the logs when the module starts up.

For instructions on how to update your device, see [Update the IoT Edge security daemon and runtime](#).

Verify the installation of IoT Edge on your devices

You can verify the installation of IoT Edge on your devices by [monitoring the edgeAgent module twin](#).

To get the latest edgeAgent module twin, run the following command from [Azure Cloud Shell](#):

Azure CLI

```
az iot hub module-twin show --device-id <edge_device_id> --module-id  
'$edgeAgent' --hub-name <iot_hub_name>
```

This command outputs all the edgeAgent [reported properties](#). Here are some helpful ones monitor the status of the device:

- runtime status
- runtime start time
- runtime last exit time
- runtime restart count

Check the status of the IoT Edge security manager and its logs

The [IoT Edge security manager](#) is responsible for operations like initializing the IoT Edge system at startup and provisioning devices. If IoT Edge isn't starting, the security manager logs may provide useful information.

- View the status of the IoT Edge system services:

Bash

```
sudo iotedge system status
```

- View the logs of the IoT Edge system services:

Bash

```
sudo iotedge system logs -- -f
```

- Enable debug-level logs to view more detailed logs of the IoT Edge system services:
 1. Enable debug-level logs.

Bash

```
sudo iotedge system set-log-level debug  
sudo iotedge system restart
```

2. Switch back to the default info-level logs after debugging.

Bash

```
sudo iotedge system set-log-level info  
sudo iotedge system restart
```

Check container logs for issues

Once the IoT Edge security daemon is running, look at the logs of the containers to detect issues. Start with your deployed containers, then look at the containers that make up the IoT Edge runtime: edgeAgent and edgeHub. The IoT Edge agent logs typically provide info on the lifecycle of each container. The IoT Edge hub logs provide info on messaging and routing.

You can retrieve the container logs from several places:

- On the IoT Edge device, run the following command to view logs:

Windows Command Prompt

```
iotedge logs <container name>
```

- On the Azure portal, use the built-in troubleshoot tool. [Monitor and troubleshoot IoT Edge devices from the Azure portal](#)
- Use the [UploadModuleLogs direct method](#) to upload the logs of a module to Azure Blob Storage.

Clean up container logs

By default the Moby container engine doesn't set container log size limits. Over time extensive logs can lead to the device filling up with logs and running out of disk space. If large container logs are affecting your IoT Edge device performance, use the following command to force remove the container along with its related logs.

If you're still troubleshooting, wait until after you've inspected the container logs to take this step.

Warning

If you force remove the edgeHub container while it has an undelivered message backlog and no [host storage](#) set up, the undelivered messages will be lost.

Windows Command Prompt

```
docker rm --force <container name>
```

For ongoing logs maintenance and production scenarios, [Set up default logging driver](#).

View the messages going through the IoT Edge hub

You can view the messages going through the IoT Edge hub and gather insights from verbose logs from the runtime containers. To turn on verbose logs on these containers, set the `RuntimeLogLevel` environment variable in the deployment manifest.

To view messages going through the IoT Edge hub, set the `RuntimeLogLevel` environment variable to `debug` for the edgeHub module.

Both the edgeHub and edgeAgent modules have this runtime log environment variable, with the default value set to `info`. This environment variable can take the following values:

- fatal
- error
- warning
- info
- debug
- verbose

You can also check the messages being sent between IoT Hub and IoT devices. View these messages by using the [Azure IoT Hub extension for Visual Studio Code](#). For more information, see [Handy tool when you develop with Azure IoT](#).

Restart containers

After investigating the logs and messages for information, you can try restarting containers.

On the IoT Edge device, use the following commands to restart modules:

```
Windows Command Prompt
```

```
iotedge restart <container name>
```

Restart the IoT Edge runtime containers:

```
Windows Command Prompt
```

```
iotedge restart edgeAgent && iotedge restart edgeHub
```

You can also restart modules remotely from the Azure portal. For more information, see [Monitor and troubleshoot IoT Edge devices from the Azure portal](#).

Check your firewall and port configuration rules

Azure IoT Edge allows communication from an on-premises server to Azure cloud using supported IoT Hub protocols. For more information, see [choosing a communication protocol](#). For enhanced security, communication channels between Azure IoT Edge and Azure IoT Hub are always configured to be Outbound. This configuration is based on the [Services Assisted Communication pattern](#), which minimizes the attack surface for a malicious entity to explore. Inbound communication is only required for [specific scenarios](#) where Azure IoT Hub needs to push messages to the Azure IoT Edge device. Cloud-to-device messages are protected using secure TLS channels and can be further secured using X.509 certificates and TPM device modules. The Azure IoT Edge Security Manager governs how this communication can be established, see [IoT Edge Security Manager](#).

While IoT Edge provides enhanced configuration for securing Azure IoT Edge runtime and deployed modules, it's still dependent on the underlying machine and network configuration. Hence, it's imperative to ensure proper network and firewall rules are set up for secure edge to cloud communication. The following table can be used as a guideline when configuration firewall rules for the underlying servers where Azure IoT Edge runtime is hosted:

[+] [Expand table](#)

| Protocol | Port | Incoming | Outgoing | Guidance |
|----------|------|----------------------|----------------------|--|
| MQTT | 8883 | BLOCKED
(Default) | BLOCKED
(Default) | <ul style="list-style-type: none"> Configure Outgoing (Outbound) to be Open when using MQTT as the communication protocol. 1883 for MQTT isn't supported by IoT Edge. Incoming (Inbound) connections should be blocked. |
| AMQP | 5671 | BLOCKED
(Default) | OPEN
(Default) | <ul style="list-style-type: none"> Default communication protocol for IoT Edge. Must be configured to be Open if Azure IoT Edge isn't configured for other supported protocols or AMQP is the desired communication protocol. 5672 for AMQP isn't supported by IoT Edge. Block this port when Azure IoT Edge uses a different IoT Hub supported protocol. Incoming (Inbound) connections should be blocked. |
| HTTPS | 443 | BLOCKED
(Default) | OPEN
(Default) | <ul style="list-style-type: none"> Configure Outgoing (Outbound) to be Open on 443 for IoT Edge provisioning. This configuration is required when using manual scripts or Azure IoT Device Provisioning Service (DPS). Incoming (Inbound) connection should be Open only for specific scenarios: <ul style="list-style-type: none"> If you have a transparent gateway with downstream devices that may send method requests. In this case, Port 443 doesn't need to be open to external networks to connect to IoTHub or provide IoTHub services through Azure IoT Edge. Thus the incoming rule could be restricted to only open Incoming (Inbound) from the internal network. For Client to Device (C2D) scenarios. 80 for HTTP isn't supported by IoT Edge. If non-HTTP protocols (for example, AMQP or MQTT) can't be configured in the enterprise; the messages can be sent over WebSockets. Port 443 is used for WebSocket communication in that case. |

Last resort: stop and recreate all containers

Sometimes, a system might require significant special modification to work with existing networking or operating system constraints. For example, a system could require a different data disk mount and proxy settings. If you tried all previous steps and still get container failures, the docker system caches or persisted network settings might not be up to date with the latest reconfiguration. In this case, the last resort option is to use [docker prune](#) to get a clean start from scratch.

The following command stops the IoT Edge system (and thus all containers), uses the "all" and "volume" option for `docker prune` to remove all containers and volumes. Review the warning that the command issues and confirm with `y` when ready.

Bash

```
sudo iotedge system stop  
docker system prune --all --volumes
```

Output

```
WARNING! This will remove:  
- all stopped containers  
- all networks not used by at least one container  
- all volumes not used by at least one container  
- all images without at least one container associated to them  
- all build cache
```

```
Are you sure you want to continue? [y/N]
```

Start the system again. To be safe, apply any potentially remaining configuration and start the system with one command.

Bash

```
sudo iotedge config apply
```

Wait a few minutes and check again.

Bash

```
sudo iotedge list
```

Next steps

Do you think that you found a bug in the IoT Edge platform? [Submit an issue ↗](#) so that we can continue to improve.

If you have more questions, create a [Support request ↗](#) for help.

Troubleshoot IoT Edge devices from the Azure portal

Article • 04/09/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

IoT Edge provides a streamlined way of monitoring and troubleshooting modules in the Azure portal. The troubleshooting page is a wrapper for the IoT Edge agent's direct methods so that you can easily retrieve logs from deployed modules and remotely restart them. This article shows you how to access and filter device and module logs in the Azure portal.

Access the troubleshooting page

You can access the troubleshooting page in the portal through either the IoT Edge device details page or the IoT Edge module details page.

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. In the left pane, select **Devices** under the **Device management** menu.
3. Select the IoT Edge device that you want to monitor from the list of devices.
4. On this device details page, you can either select **Troubleshoot** from the menu.

The screenshot shows the Azure IoT Hub Device Details page for a device named 'my-device'. The top navigation bar includes options like 'Save', 'Set modules', 'Manage child devices', 'Troubleshoot' (which is highlighted with a red box), 'Device twin', and 'Refresh'. Below the navigation, there are several configuration fields:

- Device ID: my-device
- Primary key: [REDACTED]
- Secondary key: [REDACTED]
- Primary connection string: [REDACTED]
- Secondary connection string: [REDACTED]
- IoT Edge runtime response: 200 -- OK
- Tags (edit): No tags
- Enable connection to IoT Hub: Enable Disable
- Parent device: No parent device

Or, select the runtime status of a particular module that you want to inspect.

The screenshot shows the Azure IoT Edge Modules page. It lists four modules with their runtime status:

| Name | Type | Specified in Deployment | Reported by Device | Runtime Status |
|----------------------------|------------------------|-------------------------|--------------------|----------------|
| \$edgeAgent | IoT Edge System Module | ✓ Yes | ✓ Yes | running |
| \$edgeHub | IoT Edge System Module | ✓ Yes | ✓ Yes | running |
| SimulatedTemperatureSensor | IoT Edge Custom Module | ✓ Yes | ✓ Yes | running |
| winonastreamanalytics | IoT Edge Custom Module | ✓ Yes | ✓ Yes | running |

A magnifying glass icon is located at the bottom right of the table area.

- From the device details page, you can also select the name of a module to open the module details page. From there, you can select **Troubleshoot** from the menu.

Home > my-device >

IoT Edge Module Details

SimulatedTemperatureSensor

Module Identity Twin Direct method Refresh Troubleshoot

Module Identity Name: my-device/SimulatedTemperatureSensor

Primary key:

Secondary key:

Connection string (primary key):

Connection string (secondary key):

View module logs in the portal

On the **Troubleshoot** page of your device, you can view and download logs from any of the running modules on your IoT Edge device.

This page has a maximum limit of 1,500 log lines, and any logs longer are truncated. If the logs are too large, the attempt to get module logs fails. In that case, try to change the time range filter to retrieve less data or consider using direct methods to [Retrieve logs from IoT Edge deployments](#) to gather larger log files.

Use the dropdown menu to choose which module to inspect.

Home > the-iot-hub | Devices > my-device >

Troubleshoot

the-iot-hub

Restart \$edgeAgent Refresh Download

\$edgeAgent \$edgeAgent \$edgeHub SimulatedTemperatureSensor

Time range: Since 15 minutes Find: Not specified

```
<6> 2023-03-15 00:54:52.278 +00:00 [INF] - Received 20 bytes of logs for SimulatedTemperatureSensor
<6> 2023-03-15 00:54:52.278 +00:00 [INF] - Successfully handled request GetModuleLogs
<6> 2023-03-15 00:55:12.115 +00:00 [INF] - Received direct method call - ping
<6> 2023-03-15 00:55:12.115 +00:00 [INF] - Received request ping with payload
<6> 2023-03-15 00:55:12.115 +00:00 [INF] - Successfully handled request ping
<6> 2023-03-15 00:55:12.225 +00:00 [INF] - Received direct method call - GetModuleLogs
<6> 2023-03-15 00:55:12.226 +00:00 [INF] - Received request GetModuleLogs with payload
<6> 2023-03-15 00:55:12.226 +00:00 [INF] - Processing request to get logs for {"schemaVersion":"1.0","items":{"id":"\\bSimulatedTemperatureSensor","tail":1500,"since":"15m","until":null,"loglevel":null,"regex":""}}, "encoding":1,"contentType":1}
g streaming logs for SimulatedTemperatureSensor
```

By default, this page displays the last 15 minutes of logs. Select the **Time range** filter to see different logs. Use the slider to select a time window within the last 60 minutes, or check **Enter time instead** to choose a specific datetime window.

Home > my-device >

Troubleshoot

the-iot-hub

Restart \$edgeAgent Refresh Download

\$edgeAgent

Time range: Since 15 minutes Find: Not specified

Time range

Since 15 minutes

0 15

Enter time instead

Since * Tue Mar 14 2023 Hour * Minutes * Seconds *
00 00 00

Until * Wed Mar 15 2023 Hour * Minutes * Seconds *
00 00 00

Apply Cancel

```
<6> 2023-03-15 01:11:02.910 +00:00 [INF] - Received
<6> 2023-03-15 01:11:02.910 +00:00 [INF] - Received
<6> 2023-03-15 01:11:02.910 +00:00 [INF] - Successfully
<6> 2023-03-15 01:11:03.024 +00:00 [INF] - Received
<6> 2023-03-15 01:11:03.025 +00:00 [INF] - Received
<6> 2023-03-15 01:11:03.025 +00:00 [INF] - Processin
{"tail":1500,"since":"15m","until":null,"loglevel":null,"rec
<6> 2023-03-15 01:11:03.038 +00:00 [INF] - Initiating
<6> 2023-03-15 01:11:03.039 +00:00 [INF] - Received
<6> 2023-03-15 01:11:03.039 +00:00 [INF] - Successfu
<6> 2023-03-15 01:12:57.310 +00:00 [INF] - Starting c
<6> 2023-03-15 01:13:00.483 +00:00 [INF] - Starting p
<6> 2023-03-15 01:13:00.483 +00:00 [INF] - Scraping
<6> 2023-03-15 01:13:00.483 +00:00 [INF] - Scraping
<6> 2023-03-15 01:13:00.487 +00:00 [INF] - Scraping
<6> 2023-03-15 01:13:00.492 +00:00 [INF] - Storing N
<6> 2023-03-15 01:13:00.495 +00:00 [INF] - Scrapped a
<6> 2023-03-15 01:13:00.496 +00:00 [INF] - Successfu
<6> 2023-03-15 01:13:01.433 +00:00 [INF] - Starting periodic operation refresh twin coming...
```

Once you have the logs and set your time filter from the module you want to troubleshoot, you can use the **Find** filter to retrieve specific lines from the logs. You can filter for either warnings or errors, or provide a specific term or phrase to search for.

The screenshot shows the 'Troubleshoot' page for a device named 'my-device'. At the top, there are buttons for 'Restart \$edgeAgent', 'Refresh', and 'Download'. Below these are dropdown menus for 'Time range' (set to 'Since 15 minutes') and 'Find' (set to 'Not specified'). A red box highlights the 'Find' input field. Below the input fields is a list of log entries. On the right side of the log list, there is a 'Find' interface with a text input field containing '.NET regular expressions such as '.*failed.*'' and two checkboxes: 'Warnings' (checked) and 'Errors' (checked). A red box highlights the 'Apply' button. At the bottom of the log list, there is a message: '<6> 2023-03-15 01:11:02.910 +00:00 [{"tail":1500,"since":"15m","until":null,"lo'.

The **Find** feature supports plaintext searches or [.NET regular expressions](#) for more complex searches.

You can download the module logs as a text file. The downloaded log file reflects any active filters you applied to the logs.

💡 Tip

The CPU utilization on a device spikes temporarily as it gathers logs in response to a request from the portal. This behavior is expected, and the utilization should stabilize after the task is complete.

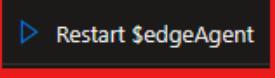
Restart modules

The **Troubleshoot** page includes a feature to restart a module. Selecting this option sends a command to the IoT Edge agent to restart the selected module. Restarting a module won't affect your ability to retrieve logs from before the restart.

Home > my-device >

Troubleshoot

the-iot-hub

 Restart \$edgeAgent  Refresh  Download

\$edgeAgent  Time range: Since 15 minutes Find: Not specified

```
<6> 2023-03-15 01:11:02.910 +00:00 [INF] - Received direct method call - ping
<6> 2023-03-15 01:11:02.910 +00:00 [INF] - Received request ping with payload
<6> 2023-03-15 01:11:02.910 +00:00 [INF] - Successfully handled request ping
<6> 2023-03-15 01:11:03.024 +00:00 [INF] - Received direct method call - GetModuleLogs
```

Next steps

Find more tips for [Troubleshooting your IoT Edge device](#) or learn about [Common issues and resolutions](#).

If you have more questions, create a [Support request](#) for help.

Solutions to common issues for Azure IoT Edge

Article • 07/10/2024

⊗ Caution

This article references CentOS, a Linux distribution that is End Of Life (EOL) status. Please consider your use and planning accordingly. For more information, see the [CentOS End Of Life guidance](#).

Applies to:  IoT Edge 1.5  IoT Edge 1.4

ⓘ Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Use this article to identify and resolve common issues when using IoT Edge solutions. If you need information on how to find logs and errors from your IoT Edge device, see [Troubleshoot your IoT Edge device](#).

Provisioning and Deployment

IoT Edge module deploys successfully then disappears from device

Symptoms

After setting modules for an IoT Edge device, the modules are deployed successfully but after a few minutes they disappear from the device and from the device details in the Azure portal. Other modules than the ones defined might also appear on the device.

Cause

If an automatic deployment targets a device, it takes priority over manually setting the modules for a single device. The **Set modules** functionality in Azure portal or **Create**

deployment for single device functionality in Visual Studio Code takes effect for a moment. You see the modules that you defined start on the device. Then the automatic deployment's priority starts and overwrites the device's desired properties.

Solution

Only use one type of deployment mechanism per device, either an automatic deployment or individual device deployments. If you have multiple automatic deployments targeting a device, you can change priority or target descriptions to make sure the correct one applies to a given device. You can also update the device twin to no longer match the target description of the automatic deployment.

For more information, see [Understand IoT Edge automatic deployments for single devices or at scale](#).

IoT Edge runtime

IoT Edge agent stops after a minute

Symptoms

The *edgeAgent* module starts and runs successfully for about a minute, then stops. The logs indicate that the IoT Edge agent attempts to connect to IoT Hub over AMQP, and then attempts to connect using AMQP over WebSocket. When that fails, the IoT Edge agent exits.

Example *edgeAgent* logs:

```
Output

2017-11-28 18:46:19 [INF] - Starting module management agent.
2017-11-28 18:46:19 [INF] - Version - 1.0.7516610
(03c94f85d0833a861a43c669842f0817924911d5)
2017-11-28 18:46:19 [INF] - Edge agent attempting to connect to IoT Hub via
AMQP...
2017-11-28 18:46:49 [INF] - Edge agent attempting to connect to IoT Hub via
AMQP over WebSocket...
```

Cause

A networking configuration on the host network is preventing the IoT Edge agent from reaching the network. The agent attempts to connect over AMQP (port 5671) first. If the

connection fails, it tries WebSockets (port 443).

The IoT Edge runtime sets up a network for each of the modules to communicate on. On Linux, this network is a bridge network. On Windows, it uses NAT. This issue is more common on Windows devices using Windows containers that use the NAT network.

Solution

Ensure that there's a route to the internet for the IP addresses assigned to this bridge/NAT network. Sometimes a VPN configuration on the host overrides the IoT Edge network.

Edge Agent module reports 'empty config file' and no modules start on the device

Symptoms

- The device has trouble starting modules defined in the deployment. Only the *edgeAgent* is running but and reports *empty config file*....
- When you run `sudo iotedge check` on a device, it reports *Container engine is not configured with DNS server setting, which may impact connectivity to IoT Hub.* Please see <https://aka.ms/iotedge-prod-checklist-dns> for best practices.

Cause

- By default, IoT Edge starts modules in their own isolated container network. The device may be having trouble with DNS name resolution within this private network.
- If using a snap installation of IoT Edge, the Docker configuration file is a different location. See solution option 3.

Solution

Option 1: Set DNS server in container engine settings

Specify the DNS server for your environment in the container engine settings, which apply to all container modules started by the engine. Create a file named `daemon.json`, then specify the DNS server to use. For example:

JSON

```
{  
  "dns": ["1.1.1.1"]  
}
```

This DNS server is set to a publicly accessible DNS service. However some networks, such as corporate networks, have their own DNS servers installed and won't allow access to public DNS servers. Therefore, if your edge device can't access a public DNS server, replace it with an accessible DNS server address.

Place `daemon.json` in the `/etc/docker` directory on your device.

If the location already contains a `daemon.json` file, add the `dns` key to it and save the file.

Restart the container engine for the updates to take effect.

Bash

```
sudo systemctl restart docker
```

Option 2: Set DNS server in IoT Edge deployment per module

You can set DNS server for each module's `createOptions` in the IoT Edge deployment. For example:

JSON

```
"createOptions": {  
  "HostConfig": {  
    "Dns": [  
      "x.x.x.x"  
    ]  
  }  
}
```

⚠ Warning

If you use this method and specify the wrong DNS address, `edgeAgent` loses connection with IoT Hub and can't receive new deployments to fix the issue. To resolve this issue, you can reinstall the IoT Edge runtime. Before you install a new instance of IoT Edge, be sure to remove any `edgeAgent` containers from the previous installation.

Be sure to set this configuration for the `edgeAgent` and `edgeHub` modules as well.

Option 3: Pass the location of the docker configuration file to check command

If IoT Edge is installed as a snap, use the `--container-engine-config-file` parameter to specify the location of the Docker configuration file. For example, if the Docker configuration file is located at `/var/snap/docker/current/config/daemon.json`, run the following command: `iotedge check --container-engine-config-file '/var/snap/docker/current/config/daemon.json'`.

Currently, the warning message continues to appear in the output of `iotedge check` even after you've set the configuration file location. Check reports the error because the IoT Edge snap doesn't have read access to the Docker snap. If you use `iotedge check` in your release process, you can suppress the warning message by using the `--ignore container-engine-dns container-engine-logrotate` parameter.

Edge Agent module with LTE connection reports 'empty edge agent config' and causes 'transient network error'

Symptoms

A device configured with LTE connection is having issues starting modules defined in the deployment. The `edgeAgent` isn't able to connect to the IoT Hub and reports *empty edge agent config* and *transient network error occurred*.

Cause

Some networks have packet overhead, which makes the default docker network MTU (1500) too high and causes packet fragmentation preventing access to external resources.

Solution

1. Check the MTU setting for your docker network.

```
docker network inspect <network name>
```

2. Check the MTU setting for the physical network adaptor on your device.

```
ip addr show eth0
```

ⓘ Note

The MTU for the docker network cannot be higher than the MTU for your device. Contact your ISP for more information.

If you see a different MTU size for your docker network and the device, try the following workaround:

1. Create a new network. For example,

```
docker network create --opt com.docker.network.driver.mtu=1430 test-mtu
```

In the example, the MTU setting for the device is 1430. Hence, the MTU for the Docker network is set to 1430.

2. Stop and remove the Azure network.

```
docker network rm azure-iot-edge
```

3. Recreate the Azure network.

```
docker network create --opt com.docker.network.driver.mtu=1430 azure-iot-edge
```

4. Remove all containers and restart the *aziot-edged* service.

```
sudo iotedge system stop && sudo docker rm -f $(docker ps -aq -f  
"label=net.azure-devices.edge.owner=Microsoft.Azure.Devices.Edge.Agent") &&  
sudo iotedge config apply
```

IoT Edge agent can't access a module's image (403)

Symptoms

A container fails to run, and the *edgeAgent* logs report a 403 error.

Cause

The IoT Edge agent module doesn't have permissions to access a module's image.

Solution

Make sure that your container registry credentials are correct your device deployment manifest.

IoT Edge agent makes excessive identity calls

Symptoms

IoT Edge agent makes excessive identity calls to Azure IoT Hub.

Cause

Device deployment manifest misconfiguration causes an unsuccessful deployment on the device. IoT Edge Agent retry logic continues to retry deployment. Each retry makes identity calls until the deployment is successful. For example, if the deployment manifest specifies a module URI that doesn't exist in the container registry or is mistyped, the IoT Edge agent retries the deployment until the deployment manifest is corrected.

Solution

Verify the deployment manifest in the Azure portal. Correct any errors and redeploy the manifest to the device.

IoT Edge hub fails to start

Symptoms

The edgeHub module fails to start. You may see a message like one of the following errors in the logs:

Output

```
One or more errors occurred.  
(Docker API responded with status code=InternalServerError, response=  
{\"message\"::\"driver failed programming external connectivity on endpoint  
edgeHub (6a82e5e994bab5187939049684fb64efe07606d2bb8a4cc5655b2a9bad5f8c80):  
Error starting userland proxy: Bind for 0.0.0.0:443 failed: port is already  
allocated\"}\n)
```

Or

Output

```
info: edgelet_docker::runtime -- Starting module edgeHub...  
warn: edgelet_utils::logging -- Could not start module edgeHub  
warn: edgelet_utils::logging -- caused by: failed to create endpoint  
edgeHub on network nat: hnsCall failed in Win32:
```

The process cannot access the file because it is being used by another process. (0x20)

Cause

Some other process on the host machine has bound a port that the edgeHub module is trying to bind. The IoT Edge hub maps ports 443, 5671, and 8883 for use in gateway scenarios. The module fails to start if another process has already bound one of those ports.

Solution

You can resolve this issue two ways:

If the IoT Edge device is functioning as a gateway device, then you need to find and stop the process that is using port 443, 5671, or 8883. An error for port 443 usually means that the other process is a web server.

If you don't need to use the IoT Edge device as a gateway, then you can remove the port bindings from edgeHub's module create options. You can change the create options in the Azure portal or directly in the deployment.json file.

In the Azure portal:

1. Navigate to your IoT hub and select **Devices** under the **Device management** menu.
2. Select the IoT Edge device that you want to update.
3. Select **Set Modules**.
4. Select **Runtime Settings**.
5. In the **Edge Hub** module settings, delete everything from the **Container Create Options** text box.
6. Select **Apply** to save your changes and create the deployment.

In the deployment.json file:

1. Open the deployment.json file that you applied to your IoT Edge device.
2. Find the `edgeHub` settings in the `edgeAgent` desired properties section:

JSON

```
"edgeHub": {  
    "restartPolicy": "always",  
    "settings": {  
        "image": "mcr.microsoft.com/azureiotedge-hub:1.5",  
        "createOptions": "{\"HostConfig\":{\"PortBindings\":[  
            {\"443/tcp\":[{\"HostPort\":443}],\"5671/tcp\":[  
                {\"HostPort\":5671}],\"8883/tcp\":[{\"HostPort\":8883}]}]}"  
    },  
    "status": "running",  
    "type": "docker"  
}
```

3. Remove the `createOptions` line, and the trailing comma at the end of the `image` line before it:

JSON

```
"edgeHub": {  
    "restartPolicy": "always",  
    "settings": {  
        "image": "mcr.microsoft.com/azureiotedge-hub:1.5",  
        "status": "running",  
        "type": "docker"  
    }  
}
```

4. Select **Create** to apply it to your IoT Edge device again.

IoT Edge module fails to send a message to edgeHub with 404 error

Symptoms

A custom IoT Edge module fails to send a message to the IoT Edge hub with a 404 `Module not found` error. The IoT Edge runtime prints the following message to the logs:

Output

```
Error: Time:Thu Jun  4 19:44:58 2018  
File:/usr/sdk/src/c/provisioning_client/adapters/hsm_client_http_edge.c  
Func:on_edge_hsm_http_recv Line:364 executing HTTP request fails,  
status=404, response_buffer={"message":"Module not found"}u, 04 )
```

Cause

The IoT Edge runtime enforces process identification for all modules connecting to the edgeHub for security reasons. It verifies that all messages being sent by a module come from the main process ID of the module. If a message is being sent by a module from a different process ID than initially established, it rejects the message with a 404 error message.

Solution

As of version 1.0.7, all module processes are authorized to connect. For more information, see the [1.0.7 release changelog ↗](#).

If upgrading to 1.0.7 isn't possible, complete the following steps. Make sure that the same process ID is always used by the custom IoT Edge module to send messages to the edgeHub. For instance, make sure to `ENTRYPOINT` instead of `CMD` command in your Docker file. The `CMD` command leads to one process ID for the module and another process ID for the bash command running the main program, but `ENTRYPOINT` leads to a single process ID.

Stability issues on smaller devices

Symptoms

You may experience stability problems on resource constrained devices like the Raspberry Pi, especially when used as a gateway. Symptoms include out of memory exceptions in the IoT Edge hub module, downstream devices failing to connect, or the device failing to send telemetry messages after a few hours.

Cause

The IoT Edge hub, which is part of the IoT Edge runtime, is optimized for performance by default and attempts to allocate large chunks of memory. This optimization isn't ideal for constrained edge devices and can cause stability problems.

Solution

For the IoT Edge hub, set an environment variable `OptimizeForPerformance` to `false`. There are two ways to set environment variables:

In the Azure portal:

1. In your IoT Hub, select your IoT Edge device and from the device details page and select **Set Modules > Runtime Settings**.
2. Create an environment variable for the IoT Edge hub module called *OptimizeForPerformance* with type *True/False* that is set to *False*.
3. Select **Apply** to save changes, then select **Review + create**.

The environment variable is now in the `edgeHub` property of the deployment manifest:

```
JSON

{
  "edgeHub": {
    "env": {
      "OptimizeForPerformance": {
        "value": false
      }
    },
    "restartPolicy": "always",
    "settings": {
      "image": "mcr.microsoft.com/azureiotedge-hub:1.5",
      "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"443/tcp\":[{\"HostPort\":\"443\"}],\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}]}}"
    },
    "status": "running",
    "type": "docker"
  }
}
```

4. Select **Create** to save your changes and deploy the module.

Security daemon couldn't start successfully

Symptoms

The security daemon fails to start and module containers aren't created. The `edgeAgent`, `edgeHub` and other custom modules aren't started by IoT Edge service. In `aziot-edged` logs, you see this error:

- The daemon could not start up successfully: Could not start management service
- caused by: An error occurred for path /var/run/iotedge/mgmt.sock
- caused by: Permission denied (os error 13)

Cause

For all Linux distros except CentOS 7, IoT Edge's default configuration is to use `systemd` socket activation. A permission error happens if you change the configuration file to not use socket activation but leave the URLs as `/var/run/iotedge/*.sock`, since the `iotedge` user can't write to `/var/run/iotedge` meaning it can't unlock and mount the sockets itself.

Solution

You don't need to disable socket activation on a distribution where socket activation is supported. However, if you prefer to not use socket activation at all, put the sockets in `/var/lib/iotedge/`.

1. Run `systemctl disable iotedge.socket iotedge.mgmt.socket` to disable the socket units so that `systemd` doesn't start them unnecessarily
2. Change the `iotedge` config to use `/var/lib/iotedge/*.sock` in both `connect` and `listen` sections
3. If you already have modules, they have the old `/var/run/iotedge/*.sock` mounts, so `docker rm -f` them.

Message queue clean up is slow

Symptoms

The message queue isn't being cleaned up after messages are processed. The message queue grows over time and eventually causes the IoT Edge runtime to run out of memory.

Cause

The message cleanup interval is controlled by the client message TTL (time to live) and the `EdgeHub MessageCleanupIntervalSecs` environment variable. The default message TTL value is two hours and the default `MessageCleanupIntervalSecs` value is 30 minutes. If your application uses a TTL value that is shorter than the default and you don't adjust the `MessageCleanupIntervalSecs` value, expired messages won't be cleaned up until the next cleanup interval.

Solution

If you change the TTL value for your application that is shorter than the default, also adjust the *MessageCleanupIntervalSecs* value. The *MessageCleanupIntervalSecs* value should be significantly smaller than the smallest TTL value that the client is using. For example, if the client application defines a TTL of five minutes in the message header, set the *MessageCleanupIntervalSecs* value to one minute. These settings ensure that messages are cleaned up within six (5 + 1) minutes.

To configure the *MessageCleanupIntervalSecs* value, set the environment variable in the deployment manifest for the IoT Edge hub module. For more information about setting runtime environment variables, see [Edge Agent and Edge Hub Environment Variables](#).

Networking

IoT Edge security daemon fails with an invalid hostname

Symptoms

Attempting to [check the IoT Edge security manager logs](#) fails and prints the following message:

Output

```
Error parsing user input data: invalid hostname. Hostname cannot be empty or greater than 64 characters
```

Cause

The IoT Edge runtime can only support hostnames that are shorter than 64 characters. Physical machines usually don't have long hostnames, but the issue is more common on a virtual machine. The automatically generated hostnames for Windows virtual machines hosted in Azure, in particular, tend to be long.

Solution

When you see this error, you can resolve it by configuring the DNS name of your virtual machine, and then setting the DNS name as the hostname in the setup command.

1. In the Azure portal, navigate to the overview page of your virtual machine.
2. Open the configuration panel by selecting **Not configured** (if your virtual machine is new) under DNS name, or select your existing DNS name. If your virtual machine

already has a DNS name configured, you don't need to configure a new one.

The screenshot shows the Azure portal interface for managing a virtual machine. At the top, there are navigation icons: Connect, Start, Restart, Stop, Capture, Delete, Refresh, and three dots. Below this, a blue bar displays an advisor message: "i Advisor (1 of 1): Machines should be configured securely →". The main content area is titled "Essentials". On the left, there's a sidebar with "Resource group (move)" set to "my-resource-group", "Status" as "Running", "Location" as "West US 3 (Zone 1)", "Subscription" set to "My subscription", "Subscription ID" as "<my-numerical-subscription-id>", and "Availability zone" as "1". On the right, details about the operating system are listed: "Operating system" is "Linux (ubuntu 20.04)". Underneath, "Size" is "Standard D2s v3 (2 vcpus, 8 GiB memory)", "Public IP address" is "19.163.15.89", "Virtual network/subnet" is "vnet-1f4d22irr522a/subnet-3m5d22irr123a", and the "DNS name" field is highlighted with a red box and contains the text "Not configured". There are also "View Cost" and "JSON View" buttons at the top right.

3. Provide a value for **DNS name label** if you don't have one already and select **Save**.
4. Copy the new DNS name, which should be in the format:
`<DNSnamelabel>.<vmlocation>.cloudapp.azure.com.`
5. On the IoT Edge device, open the config file.

```
Bash  
sudo nano /etc/aziot/config.toml
```

6. Replace the value of `hostname` with your DNS name.
7. Save and close the file, then apply the changes to IoT Edge.

```
Bash  
sudo iotedge config apply
```

IoT Edge module reports connectivity errors

Symptoms

IoT Edge modules that connect directly to cloud services, including the runtime modules, stop working as expected and return errors around connection or networking failures.

Cause

Containers rely on IP packet forwarding in order to connect to the internet so that they can communicate with cloud services. IP packet forwarding is enabled by default in Docker, but if it gets disabled then any modules that connect to cloud services won't work as expected. For more information, see [Understand container communication](#) in the Docker documentation.

Solution

Use the following steps to enable IP packet forwarding.

1. Open the `sysctl.conf` file.

```
Bash
sudo nano /etc/sysctl.conf
```

2. Add the following line to the file.

```
input
net.ipv4.ip_forward=1
```

3. Save and close the file.

4. Restart the network service and docker service to apply the changes.

IoT Edge behind a gateway can't perform HTTP requests and start edgeAgent module

Symptoms

The IoT Edge runtime is active with a valid configuration file, but it can't start the `edgeAgent` module. The command `iotedge list` returns an empty list. The IoT Edge runtime reports `Could not perform HTTP request` in the logs.

Cause

IoT Edge devices behind a gateway get their module images from the parent IoT Edge device specified in the `parent_hostname` field of the config file. The `Could not perform`

`HTTP request` error means that the downstream device isn't able to reach its parent device via HTTP.

Solution

Make sure the parent IoT Edge device can receive incoming requests from the downstream IoT Edge device. Open network traffic on ports 443 and 6617 for requests coming from the downstream device.

IoT Edge behind a gateway can't perform HTTP requests and start edgeAgent module

Symptoms

The IoT Edge daemon is active with a valid configuration file, but it can't start the `edgeAgent` module. The command `iotedge list` returns an empty list. The IoT Edge daemon logs report `Could not perform HTTP request`.

Cause

IoT Edge devices behind a gateway get their module images from the parent IoT Edge device specified in the `parent_hostname` field of the config file. The `Could not perform HTTP request` error means that the downstream device isn't able to reach its parent device via HTTP.

Solution

Make sure the parent IoT Edge device can receive incoming requests from the downstream IoT Edge device. Open network traffic on ports 443 and 6617 for requests coming from the downstream device.

IoT Edge behind a gateway can't connect when migrating from one IoT hub to another

Symptoms

When attempting to migrate a hierarchy of IoT Edge devices from one IoT hub to another, the top level parent IoT Edge device can connect to IoT Hub, but downstream

IoT Edge devices can't. The logs report `Unable to authenticate client downstream-device/$edgeAgent with module credentials.`

Cause

The credentials for the downstream devices weren't updated properly when the migration to the new IoT hub happened. Because of this, `edgeAgent` and `edgeHub` modules were set to have authentication type of `none` (default if not set explicitly). During connection, the modules on the downstream devices use old credentials, causing the authentication to fail.

Solution

When migrating to the new IoT hub (assuming not using DPS), follow these steps in order:

1. Follow [this guide to export and then import device identities](#) from the old IoT hub to the new one
2. Reconfigure all IoT Edge deployments and configurations in the new IoT hub
3. Reconfigure all parent-child device relationships in the new IoT hub
4. Update each device to point to the new IoT hub hostname (`iothub_hostname` under `[provisioning]` in `config.toml`)
5. If you chose to exclude authentication keys during the device export, reconfigure each device with the new keys given by the new IoT hub (`device_id_pk` under `[provisioning.authentication]` in `config.toml`)
6. Restart the top-level parent Edge device first, make sure it's up and running
7. Restart each device in hierarchy level by level from top to the bottom

IoT Edge has low message throughput when geographically distant from IoT Hub

Symptoms

Azure IoT Edge devices that are geographically distant from Azure IoT Hub have a lower than expected message throughput.

Cause

High latency between the device and IoT Hub can cause a lower than expected message throughput. IoT Edge uses a default message batch size of 10. This limits the number of messages that are sent in a single batch, which increases the number of round trips between the device and IoT Hub.

Solution

Try increasing the IoT Edge Hub **MaxUpstreamBatchSize** environment variable. This allows more messages to be sent in a single batch, which reduces the number of round trips between the device and IoT Hub.

To set Azure Edge Hub environment variables in the Azure portal:

1. Navigate to your IoT Hub and select **Devices** under the **Device management** menu.
2. Select the IoT Edge device that you want to update.
3. Select **Set Modules**.
4. Select **Runtime Settings**.
5. In the **Edge Hub** module settings tab, add the **MaxUpstreamBatchSize** environment variable as type **Number** with a value of 20.
6. Select **Apply**.

Next steps

Do you think that you found a bug in the IoT Edge platform? [Submit an issue ↗](#) so that we can continue to improve.

If you have more questions, create a [Support request ↗](#) for help.

Feedback

Was this page helpful?



[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

What is Azure IoT Edge for Linux on Windows

Article • 06/05/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge for Linux on Windows (EFLLOW) allows you to run containerized Linux workloads alongside Windows applications in Windows deployments. Businesses that rely on Windows to power their edge devices and solutions can now take advantage of the cloud-native analytics solutions being built in Linux.

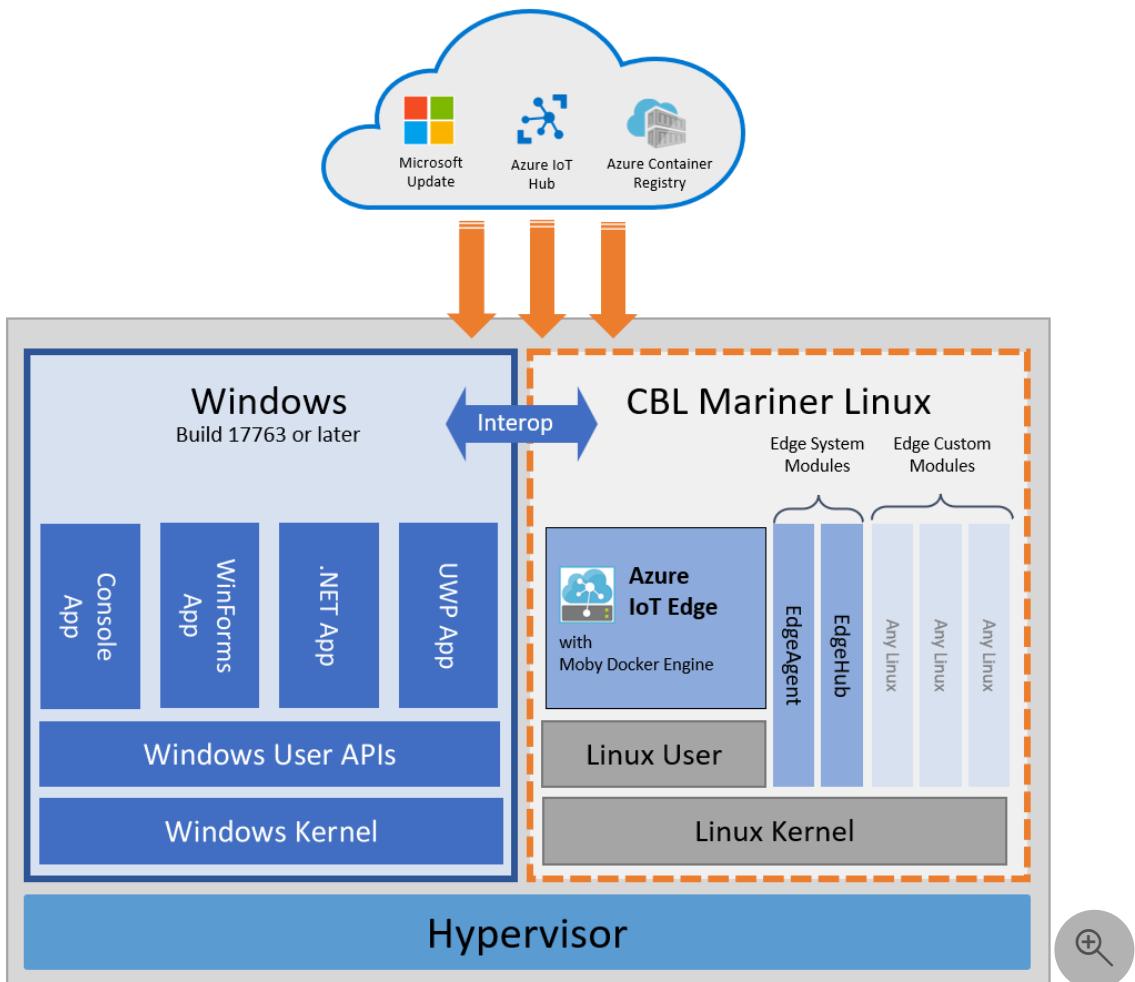
Azure IoT Edge for Linux on Windows works by running a Linux virtual machine on a Windows device. The Linux virtual machine comes pre-installed with the Azure IoT Edge runtime. Any Azure IoT Edge modules deployed to the device run inside the virtual machine. Meanwhile, Windows applications running on the Windows host device can communicate with the modules running in the Linux virtual machine.

[Get started today.](#)

Components

Azure IoT Edge for Linux on Windows uses the following components to enable Linux and Windows workloads to run alongside each other and communicate seamlessly:

- **A Linux virtual machine running Azure IoT Edge:** A Linux virtual machine, based on Microsoft's first party [CBL-Mariner](#) operating system, is built with the Azure IoT Edge runtime and validated as a tier 1 supported environment for Azure IoT Edge workloads.
- **Microsoft Update:** Integration with Microsoft Update keeps the Windows runtime components, the CBL-Mariner Linux VM, and Azure IoT Edge up to date. For more information about IoT Edge for Linux on Windows updates, see [Update IoT Edge for Linux on Windows](#).



Bi-directional communication between Windows process and the Linux virtual machine means that Windows processes can provide user interfaces or hardware proxies for workloads run in the Linux containers.

Prerequisites

A Windows device with the following minimum requirements:

- System Requirements
 - Windows 10¹/11 (Pro, Enterprise, IoT Enterprise)
 - Windows Server 2019¹/2022

¹ Windows 10 and Windows Server 2019 minimum build 17763 with all current cumulative updates installed.

- Hardware requirements
 - Minimum Free Memory: 1 GB
 - Minimum Free Disk Space: 10 GB

For more information about IoT Edge for Linux on Windows requirements, see [Azure IoT Edge for Linux on Windows supported systems](#).

Platform support

Azure IoT Edge for Linux on Windows supports both AMD64 and ARM64 architectures. For more information about EFLOW platform support, see [Azure IoT Edge for Linux on Windows supported systems](#)

Samples

Azure IoT Edge for Linux on Windows emphasizes interoperability between the Linux and Windows components.

For samples that demonstrate communication between Windows applications and Azure IoT Edge modules, see [EFLow GitHub](#).

Also, you can use your IoT Edge for Linux on Windows device to act as a transparent gateway for other edge devices. For more information on how to configure EFLOW as a transparent gateway, see [Configure an IoT Edge device to act as a transparent gateway](#).

Support

Use the Azure IoT Edge support and feedback channels to get assistance with Azure IoT Edge for Linux on Windows.

Reporting bugs - Bugs related to Azure IoT Edge for Linux on Windows can be reported on the [iotedge-eflow issues page](#). Bugs related to Azure IoT Edge can be reported on the [issues page](#) of the Azure IoT Edge open-source project.

Microsoft Customer Support team - Users who have a [support plan](#) can engage the Microsoft Customer Support team by creating a support ticket directly from the [Azure portal](#).

Feature requests - The Azure IoT Edge product tracks feature requests via the product's [User Voice page](#).

Next steps

Watch [Azure IoT Edge for Linux on Windows 10 IoT Enterprise](#) for more information and a sample in action.

Follow the steps in [Manually provision a single Azure IoT Edge for Linux on a Windows device](#) to set up a device with Azure IoT Edge for Linux on Windows.

Quickstart: Deploy your first IoT Edge module to a Windows device

Article • 07/08/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

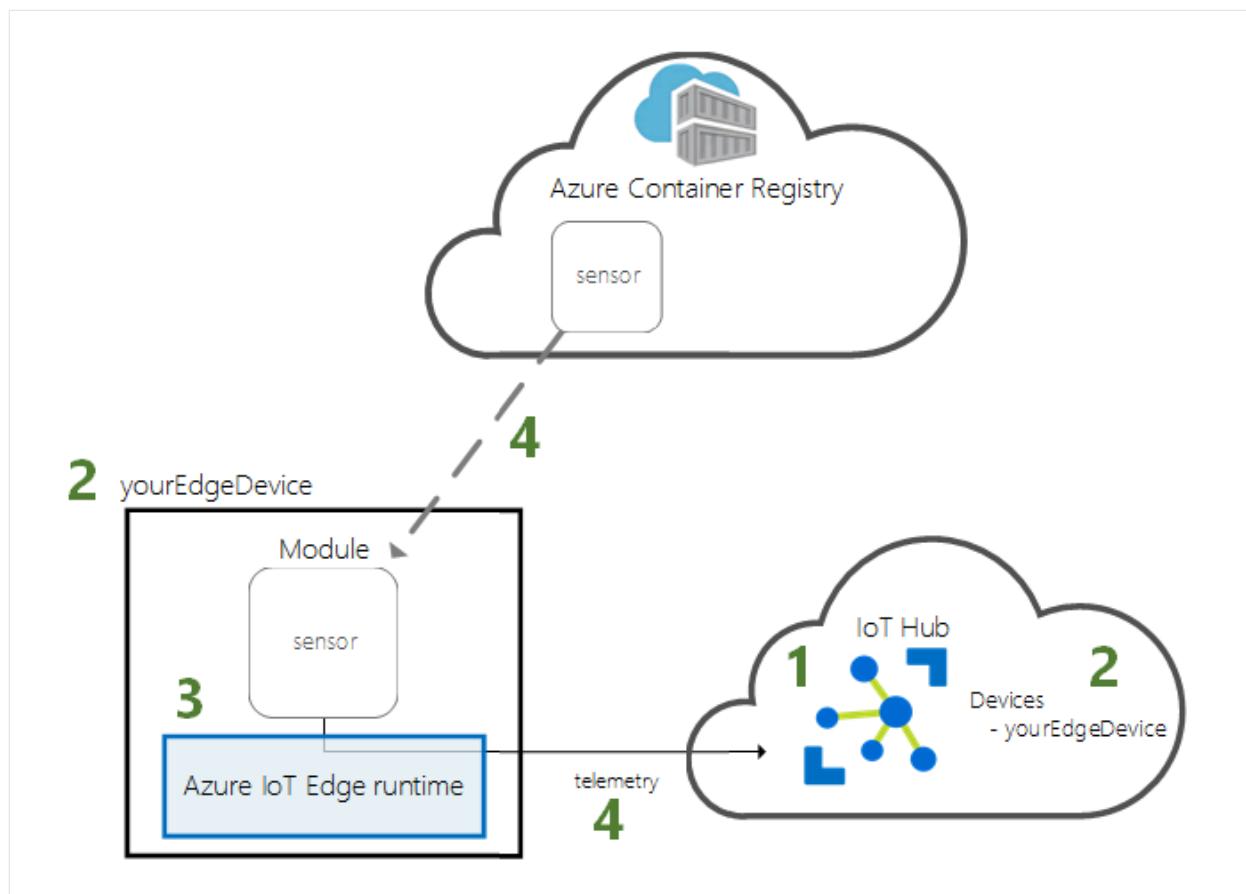
Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Try out Azure IoT Edge in this quickstart by deploying containerized code to a Linux on Windows IoT Edge device. IoT Edge allows you to remotely manage code on your devices so that you can send more of your workloads to the edge. For this quickstart, we recommend using your own Windows Client device to see how easy it is to use Azure IoT Edge for Linux on Windows. If you wish to use Windows Server or an Azure VM to create your deployment, follow the steps in the how-to guide on [installing and provisioning Azure IoT Edge for Linux on a Windows device](#).

In this quickstart, you'll learn how to:

- Create an IoT hub.
- Register an IoT Edge device to your IoT hub.
- Install and start the IoT Edge for Linux on Windows runtime on your device.
- Remotely deploy a module to an IoT Edge device and send telemetry.



This quickstart walks you through how to set up your Azure IoT Edge for Linux on Windows device. Then, you'll deploy a module from the Azure portal to your device. The module you'll use is a simulated sensor that generates temperature, humidity, and pressure data. Other Azure IoT Edge tutorials build on the work you do here by deploying modules that analyze the simulated data for business insights.

If you don't have an active Azure subscription, create a [free account](#) before you begin.

Prerequisites

Prepare your environment for the Azure CLI.

- Use the Bash environment in [Azure Cloud Shell](#). For more information, see [Quickstart for Bash in Azure Cloud Shell](#).
- [A Launch Cloud Shell](#)
- If you prefer to run CLI reference commands locally, [install](#) the Azure CLI. If you're running on Windows or macOS, consider running Azure CLI in a Docker container. For more information, see [How to run the Azure CLI in a Docker container](#).
 - If you're using a local installation, sign in to the Azure CLI by using the [az login](#) command. To finish the authentication process, follow the steps displayed in

your terminal. For other sign-in options, see [Sign in with the Azure CLI](#).

- When you're prompted, install the Azure CLI extension on first use. For more information about extensions, see [Use extensions with the Azure CLI](#).
- Run `az version` to find the version and dependent libraries that are installed. To upgrade to the latest version, run `az upgrade`.

Create a cloud resource group to manage all the resources you'll use in this quickstart.

Azure CLI

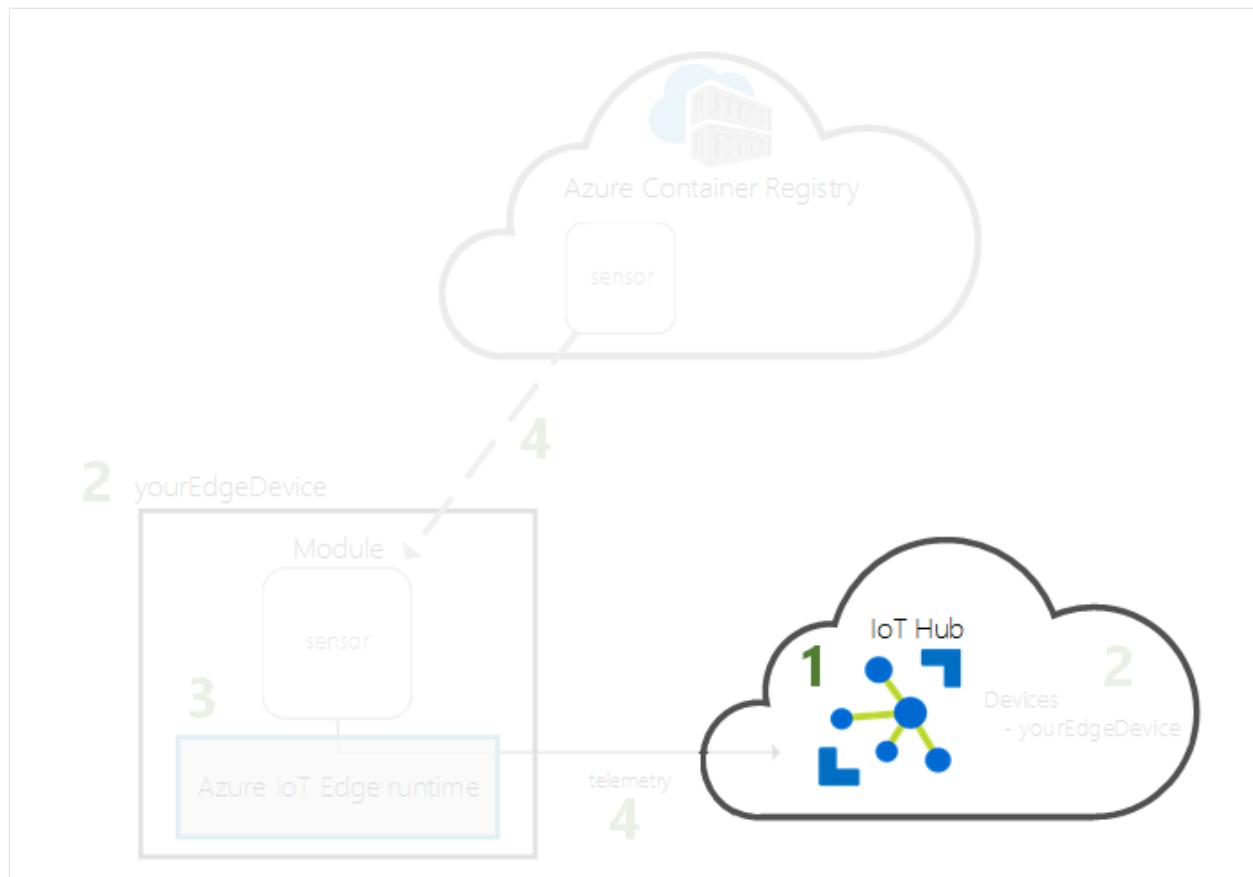
```
az group create --name IoTEdgeResources --location westus2
```

Make sure your IoT Edge device meets the following requirements:

- System Requirements
 - Windows 10¹/11 (Pro, Enterprise, IoT Enterprise) ¹ Windows 10 minimum build 17763 with all current cumulative updates installed.
- Hardware requirements
 - Minimum Free Memory: 1 GB
 - Minimum Free Disk Space: 10 GB

Create an IoT hub

Start by creating an IoT hub with the Azure CLI.



The free level of Azure IoT Hub works for this quickstart. If you've used IoT Hub in the past and already have a hub created, you can use that IoT hub.

The following code creates a free F1 hub in the resource group `IoTEdgeResources`.

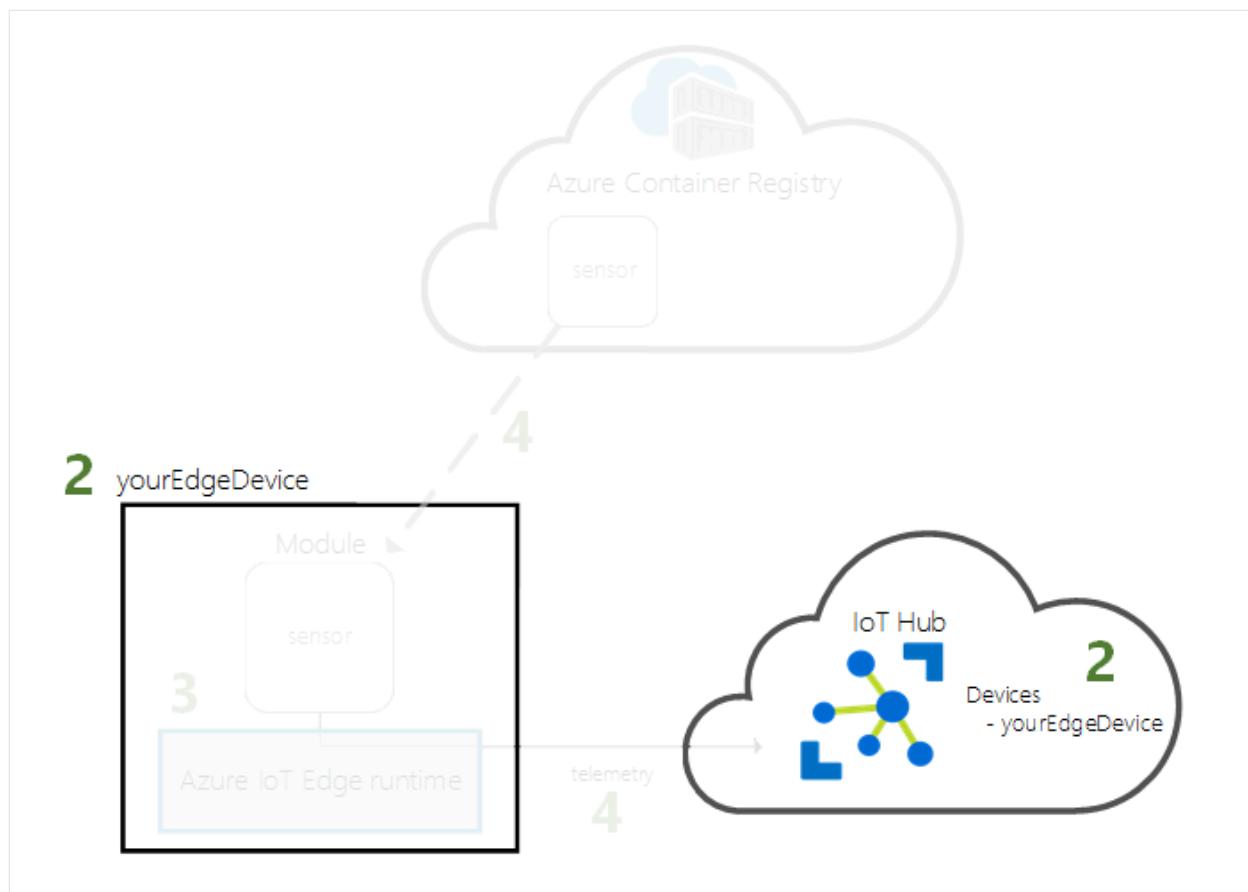
Replace `{hub_name}` with a unique name for your IoT hub. It might take a few minutes to create an IoT hub.

```
Azure CLI
az iot hub create --resource-group IoTEdgeResources --name {hub_name} --sku
F1 --partition-count 2
```

If you get an error because you already have one free hub in your subscription, change the SKU to `S1`. If you get an error that the IoT hub name isn't available, someone else already has a hub with that name. Try a new name.

Register an IoT Edge device

Register an IoT Edge device with your newly created IoT hub.



Create a device identity for your simulated device so that it can communicate with your IoT hub. The device identity lives in the cloud, and you use a unique device connection string to associate a physical device to a device identity.

IoT Edge devices behave and can be managed differently than typical IoT devices. Use the `--edge-enabled` flag to declare that this identity is for an IoT Edge device.

1. In Azure Cloud Shell, enter the following command to create a device named `myEdgeDevice` in your hub.

Azure CLI

```
az iot hub device-identity create --device-id myEdgeDevice --edge-enabled --hub-name {hub_name}
```

If you get an error about `iothubowner` policy keys, make sure that Cloud Shell is running the latest version of the Azure IoT extension.

2. View the connection string for your device, which links your physical device with its identity in IoT Hub. It contains the name of your IoT hub, the name of your device, and a shared key that authenticates connections between the two.

Azure CLI

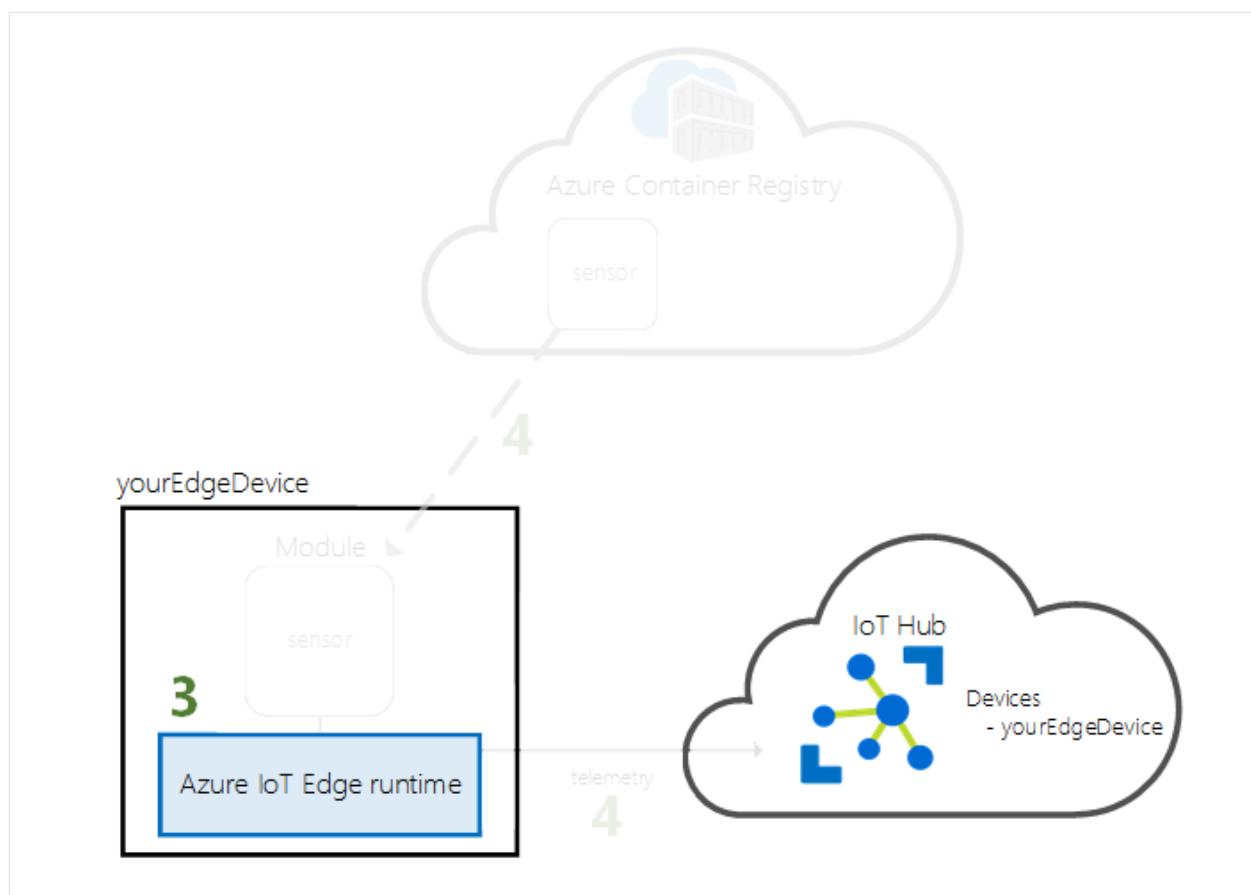
```
az iot hub device-identity connection-string show --device-id  
myEdgeDevice --hub-name {hub_name}
```

3. Copy the value of the `connectionString` key from the JSON output and save it. This value is the device connection string. You'll use it to configure the IoT Edge runtime in the next section.

For example, your connection string should look similar to `HostName=contoso-hub.azure-devices.net;DeviceId=myEdgeDevice;SharedAccessKey=<DEVICE_SHARED_ACCESS_KEY>`.

Install and start the IoT Edge runtime

Install IoT Edge for Linux on Windows on your device, and configure it with the device connection string.



Run the following PowerShell commands on the target device where you want to deploy Azure IoT Edge for Linux on Windows. To deploy to a remote target device using PowerShell, use [Remote PowerShell](#) to establish a connection to a remote device and run these commands remotely on that device.

1. In an elevated PowerShell session, run the following command to enable Hyper-V.
For more information, check [Hyper-V on Windows 10](#).

```
PowerShell
```

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Hyper-V -  
All
```

2. In an elevated PowerShell session, run each of the following commands to download IoT Edge for Linux on Windows.

- X64/AMD64

```
PowerShell
```

```
$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))  
$ProgressPreference = 'SilentlyContinue'  
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_X64" -OutFile  
$msiPath
```

- ARM64

```
PowerShell
```

```
$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))  
$ProgressPreference = 'SilentlyContinue'  
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_ARM64" -  
OutFile $msiPath
```

3. Install IoT Edge for Linux on Windows on your device.

```
PowerShell
```

```
Start-Process -Wait msieexec -ArgumentList  
"/i","$([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))","/qn"
```

4. Set the execution policy on the target device to `AllSigned` if it is not already. You can check the current execution policy in an elevated PowerShell prompt using:

```
PowerShell
```

```
Get-ExecutionPolicy -List
```

If the execution policy of `local machine` is not `AllSigned`, you can set the execution policy using:

PowerShell

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Force
```

5. Create the IoT Edge for Linux on Windows deployment.

PowerShell

```
Deploy-Eflow
```

6. Enter 'Y' to accept the license terms.

7. Enter 'O' or 'R' to toggle **Optional diagnostic data** on or off, depending on your preference. A successful deployment is pictured below.

```
- Creating storage vhd (file: AzureIoTEdgeForLinux-v1-EFLOW)
- Creating vnic (name: DESKTOP-SFRE9NQ-EFLOWInterface)
- Instantiating virtual machine (name: DESKTOP-SFRE9NQ-EFLOW)
- Virtual machine successfully instantiated

[06/16/2021 09:32:20] Virtual machine created successfully.

[06/16/2021 09:32:22] Successfully created virtual machine

[06/16/2021 09:32:22] Virtual machine hostname: DESKTOP-SFRE9NQ-EFLOW

[06/16/2021 09:32:24] Querying IP and MAC addresses from virtual machine (DESKTOP-SFRE9NQ-EFLOW)

- Virtual machine MAC: 00:15:5d:c5:64:9c
- Virtual machine IP : 172.20.101.181

[06/16/2021 09:32:28] Testing SSH connection...

[06/16/2021 09:32:38] ...successfully connected to the Linux VM

[06/16/2021 09:32:42] Deployment successful
```

8. Provision your device using the device connection string that you retrieved in the previous section. Replace the placeholder text with your own value.

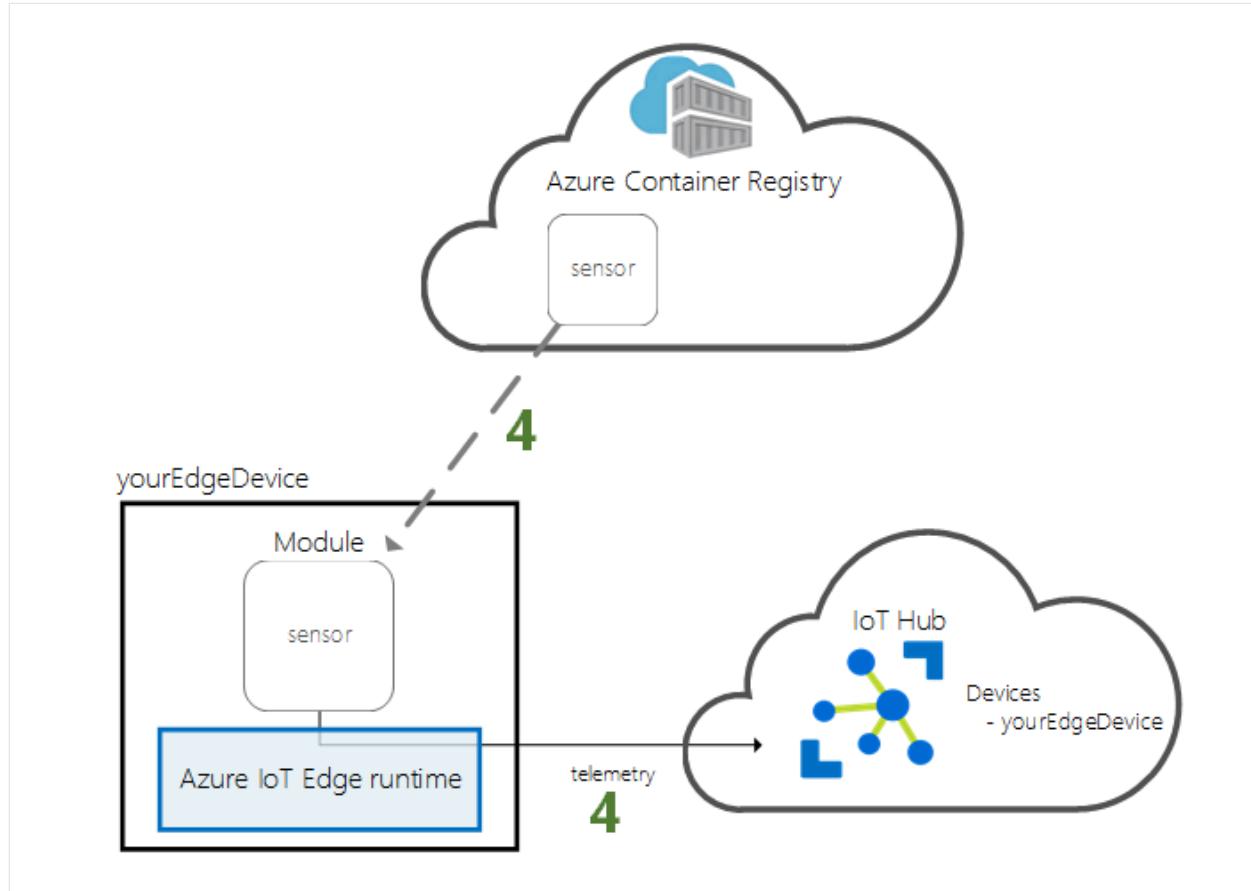
PowerShell

```
Provision-EflowVm -provisioningType ManualConnectionString -
devConnString "<CONNECTION_STRING_HERE>"
```

Your IoT Edge device is now configured. It's ready to run cloud-deployed modules.

Deploy a module

Manage your Azure IoT Edge device from the cloud to deploy a module that sends telemetry data to IoT Hub.



One of the key capabilities of Azure IoT Edge is deploying code to your IoT Edge devices from the cloud. *IoT Edge modules* are executable packages implemented as containers. In this section, you'll deploy a pre-built module from the [IoT Edge Modules section of Microsoft Artifact Registry](#).

The module that you deploy in this section simulates a sensor and sends generated data. This module is a useful piece of code when you're getting started with IoT Edge because you can use the simulated data for development and testing. If you want to see exactly what this module does, you can view the [simulated temperature sensor source code](#).

Follow these steps to deploy your first module.

1. Sign in to the [Azure portal](#) and go to your IoT Hub.
2. From the menu on the left, select **Devices** under the **Device management** menu.
3. Select the device ID of the target device from the list of devices.

! Note

When you create a new IoT Edge device, it will display the status code 417 -- The device's deployment configuration is not set in the Azure portal. This status is normal, and means that the device is ready to receive a module deployment.

4. On the upper bar, select **Set Modules**.

Choose which modules you want to run on your device. You can choose from modules that you've already created, modules from Microsoft Artifact Registry, or modules that you've built yourself. In this quickstart, you'll deploy a module from the Microsoft Artifact Registry.

5. In the **IoT Edge modules** section, select **Add** then choose **IoT Edge Module**.

6. Update the following module settings:

[\[+\] Expand table](#)

| Setting | Value |
|-----------------|--|
| IoT Module name | SimulatedTemperatureSensor |
| Image URI | mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:latest |
| Restart policy | always |
| Desired status | running |

7. Select **Next: Routes** to continue to configure routes.

8. Add a route that sends all messages from the simulated temperature module to IoT Hub.

[\[+\] Expand table](#)

| Setting | Value |
|---------|---|
| Name | SimulatedTemperatureSensorToIoTHub |
| Value | FROM /messages/modules/SimulatedTemperatureSensor/* INTO \$upstream |

9. Select **Next: Review + create**.

10. Review the JSON file, and then select **Create**. The JSON file defines all of the modules that you deploy to your IoT Edge device.

(!) Note

When you submit a new deployment to an IoT Edge device, nothing is pushed to your device. Instead, the device queries IoT Hub regularly for any new instructions. If the device finds an updated deployment manifest, it uses the information about the new deployment to pull the module images from the cloud then starts running the modules locally. This process can take a few minutes.

After you create the module deployment details, the wizard returns you to the device details page. View the deployment status on the **Modules** tab.

You should see three modules: `$edgeAgent`, `$edgeHub`, and `SimulatedTemperatureSensor`. If one or more of the modules has **Yes** under **Specified in Deployment** but not under **Reported by Device**, your IoT Edge device is still starting them. Wait a few minutes, and then refresh the page.

| Device Details | | | | | | |
|---|--------------------------|--------------------------------|--------------------|----------------|-----------|--|
| Modules | IoT Edge hub connections | Deployments and Configurations | | | | |
| Name | Type | Specified in Deployment | Reported by Device | Runtime Status | Exit Code | |
| <code>\$edgeAgent</code> | IoT Edge System Module | ✓ Yes | ✓ Yes | running | NA | |
| <code>\$edgeHub</code> | IoT Edge System Module | ✓ Yes | ✓ Yes | running | NA | |
| <code>SimulatedTemperatureSensor</code> | IoT Edge Custom Module | ✓ Yes | ✓ Yes | running | NA | |

If you have issues deploying modules, see [Troubleshoot IoT Edge devices from the Azure portal](#).

View the generated data

In this quickstart, you created a new IoT Edge device and installed the IoT Edge runtime on it. Then you used the Azure portal to deploy an IoT Edge module to run on the device without having to make changes to the device itself.

The module that you pushed generates sample environment data that you can use for testing later. The simulated sensor is monitoring both a machine and the environment around the machine. For example, this sensor might be in a server room, on a factory floor, or on a wind turbine. The messages that it sends include ambient temperature and humidity, machine temperature and pressure, and a timestamp. IoT Edge tutorials use the data created by this module as test data for analytics.

1. Log in to your IoT Edge for Linux on Windows virtual machine using the following command in your PowerShell session:

```
PowerShell
```

```
Connect-EflowVm
```

ⓘ Note

The only account allowed to SSH to the virtual machine is the user that created it.

2. Once you are logged in, you can check the list of running IoT Edge modules using the following Linux command:

```
Bash
```

```
sudo iotedge list
```

```
iotedge-user@DESKTOP-7BL390G-EFLOW-9f5817c6 [ ~ ]$ sudo iotedge list
NAME          STATUS    DESCRIPTION           CONFIG
SimulatedTemperatureSensor  running   Up 3 hours   mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.0
edgeAgent      running   Up 3 hours   mcr.microsoft.com/azureiotedge-agent:1.0
edgeHub        running   Up 3 hours   mcr.microsoft.com/azureiotedge-hub:1.0
iotedge-user@DESKTOP-7BL390G-EFLOW-9f5817c6 [ ~ ]$
```

3. View the messages being sent from the temperature sensor module to the cloud using the following Linux command:

```
Bash
```

```
sudo iotedge logs SimulatedTemperatureSensor -f
```

```
iotedge-user@DESKTOP-7BL390G-EFLOW-9f5817c6 [ ~ ]$ iotedge-user@DESKTOP-7BL390G-EFLOW-9f5817c6 [ ~ ]$ iotedge logs SimulatedTemperatureSensor -f
[2020-12-22 19:30:41 : Starting Module
SimulatedTemperatureSensor Main() started.
Initializing simulated temperature sensor to send 500 messages, at an interval of 5 seconds.
To change this, set the environment variable MessageCount to the number of messages that should be sent (set it to -1 to send unlimited messages).
[Information]: Trying to initialize module client using transport type [Ampq Tcp Only].
[Information]: Successfully initialized module client of transport type [Ampq Tcp Only].
12/22/2020 19:30:43 > Sending message: 1, Body: [{"machine": {"temperature": 22.026816424869777, "pressure": 1.116979086370734}, "ambient": {"temperature": 20.7195741223262
5, "humidity": 26}, "timeCreated": "2020-12-22T19:30:43.943895827"}]
12/22/2020 19:30:49 > Sending message: 2, Body: [{"machine": {"temperature": 21.983345918349617, "pressure": 1.1120267501917285}, "ambient": {"temperature": 21.211475789412
614, "humidity": 25}, "timeCreated": "2020-12-22T19:30:49.124673527"}]
12/22/2020 19:30:54 > Sending message: 3, Body: [{"machine": {"temperature": 22.687484067258184, "pressure": 1.1831310962699197}, "ambient": {"temperature": 21.1950622446962
944, "humidity": 24}, "timeCreated": "2020-12-22T19:30:54.161353127"}]
12/22/2020 19:30:59 > Sending message: 4, Body: [{"machine": {"temperature": 23.825498643715633, "pressure": 1.3218922565498822}, "ambient": {"temperature": 21.259362409710
587, "humidity": 25}, "timeCreated": "2020-12-22T19:30:59.188859627"}]
12/22/2020 19:31:04 > Sending message: 5, Body: [{"machine": {"temperature": 24.76149326260737, "pressure": 1.4285245489046372}, "ambient": {"temperature": 21.3238176153152
3, "humidity": 25}, "timeCreated": "2020-12-22T19:31:04.217112627"}]
12/22/2020 19:31:09 > Sending message: 6, Body: [{"machine": {"temperature": 25.801075597666706, "pressure": 1.546957979481017}, "ambient": {"temperature": 20.7239539507888
94, "humidity": 26}, "timeCreated": "2020-12-22T19:31:09.244597327"}]
12/22/2020 19:31:14 > Sending message: 7, Body: [{"machine": {"temperature": 26.2118842459339, "pressure": 1.5936796939410192}, "ambient": {"temperature": 20.6344077936999
53, "humidity": 25}, "timeCreated": "2020-12-22T19:31:14.275180527"}]
```

💡 Tip

IoT Edge commands are case-sensitive when they refer to module names.

Clean up resources

If you want to continue on to the IoT Edge tutorials, skip this step. You can use the device that you registered and set up in this quickstart. Otherwise, you can delete the Azure resources that you created to avoid charges.

If you created your virtual machine and IoT hub in a new resource group, you can delete that group and all the associated resources. If you don't want to delete the whole group, you can delete individual resources instead.

ⓘ Important

Check the contents of the resource group to make sure that there's nothing you want to keep. Deleting a resource group is irreversible.

Use the following command to remove the **IoTEdgeResources** group. Deletion might take a few minutes.

Azure CLI

```
az group delete --name IoTEdgeResources
```

You can confirm that the resource group is removed by using this command to view the list of resource groups.

Azure CLI

```
az group list
```

Uninstall IoT Edge for Linux on Windows

If you want to remove the Azure IoT Edge for Linux on Windows installation from your device, use the following commands.

1. Open Settings on Windows
2. Select Add or Remove Programs
3. Select *Azure IoT Edge* app
4. Select Uninstall

Next steps

In this quickstart, you created an IoT Edge device and used the Azure IoT Edge cloud interface to deploy code onto the device. Now you have a test device generating raw data about its environment.

In the next tutorial, you'll learn how to monitor the activity and health of your device from the Azure portal.

[Monitor IoT Edge devices](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Why use Azure IoT Edge for Linux on Windows?

Article • 06/13/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

For organizations interested in running business logic and analytics on devices, Azure IoT Edge for Linux on Windows (EFLOW) enables the deployment of production Linux-based cloud-native workloads onto Windows devices. Connecting your devices to Microsoft Azure lets you quickly bring cloud intelligence to your business. At the same time, running workloads on devices allows you to respond quickly in instances with limited connectivity and reduce bandwidth costs.

By bringing the best of Windows and Linux together, EFLOW enables new capabilities while leveraging existing Windows infrastructure and application investments. By running Linux IoT Edge modules on Windows devices, you can do more on a single device, reducing the overhead and cost of separate devices for different applications.

EFLOW doesn't require extensive Linux knowledge and utilizes familiar Windows tools to manage your EFLOW device and workloads. Windows IoT provides trusted enterprise-grade security with established IT admin infrastructure. Lastly, the entire solution is maintained and kept up to date by Microsoft.

Easily Connect to Azure

IoT Edge Built-In. [Tier 1 Azure IoT Edge support](#) is built in to EFLOW for a simplified deployment experience for your cloud workloads.

Curated Linux VM for Azure. EFLOW consists of a specially curated Linux VM that runs alongside Windows IoT host OS. This Linux VM is based on [CBL-Mariner Linux](#), and is optimized for hosting IoT Edge workloads.

Familiar Windows Management

Flexible Scripting. [PowerShell modules](#) provide the ability to fully script deployments.

WAC. [Windows Admin Center EFLOW extension](#) (preview, EFLOW 1.1 only) provides a click-through deployment wizard and remote management experience.

Production Ready

Always Up-to-date. EFLOW regularly releases feature and security improvements and is reliably updated using Microsoft Update. For more information on EFLOW updates, see [Update IoT Edge for Linux on Windows](#).

Fully Supported Environment. In an EFLOW solution, the base operating system, the EFLOW Linux environment, and the container runtime are all maintained by Microsoft—meaning there's a single source for all of the components. Each of the three components: [Windows IoT](#), EFLOW, and [Azure IoT Edge](#) have defined servicing mechanisms and support timelines.

Windows + Linux

Interoperability. With EFLOW, the whole is greater than the sum of its parts. Combining a Windows application and Linux application on the same device unlocks new experiences and scenarios that otherwise wouldn't have been possible. Interoperability and hardware passthrough capabilities built into EFLOW including, [TPM passthrough](#), [HW acceleration](#), [Camera passthrough](#), [Serial passthrough](#), and more, allow you to take advantage of both Linux and Windows environments.

Feedback

Was this page helpful?



[Provide product feedback](#)

Update IoT Edge for Linux on Windows

Article • 06/05/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

As the IoT Edge for Linux on Windows (EFLOW) application releases new versions, you'll want to update your IoT Edge devices for the latest features and security improvements. This article provides information about how to update your IoT Edge for Linux on Windows devices when a new version is available.

With IoT Edge for Linux on Windows, IoT Edge runs in a Linux virtual machine hosted on a Windows device. This virtual machine is pre-installed with IoT Edge, and has no package manager, so you can't manually update or change any of the VM components. Instead, the virtual machine is managed with Microsoft Update to keep the components up to date automatically.

The EFLOW virtual machine is designed to be reliably updated via Microsoft Update. The virtual machine operating system has an A/B update partition scheme to utilize a subset of those to make each update safe and enable a roll-back to a previous version if anything goes wrong during the update process.

Each update consists of two main components that may get updated to latest versions. The first one is the EFLOW virtual machine and the internal components. For more information about EFLOW, see [Azure IoT Edge for Linux on Windows composition](#). This also includes the virtual machine base operating system. The EFLOW virtual machine is based on [Microsoft CBL-Mariner](#) and each update provides performance and security fixes to keep the OS with the latest CVE patches. As part of the EFLOW Release notes, the version indicates the CBL-Mariner version used, and users can check the [CBL-Mariner Releases](#) to get the list of CVEs fixed for each version.

The second component is the group of Windows runtime components needed to run and interop with the EFLOW virtual machine. The virtual machine lifecycle and interop is managed through different components: WSSDAgent, EFLOWProxy service and the PowerShell module.

EFLOW updates are sequential and you'll require to update to every version in order, which means that in order to get to the latest version, you'll have to either do a fresh installation using the latest available version, or apply all the previous servicing updates up to the desired version.

To find the latest version of Azure IoT Edge for Linux on Windows, see [EFLOW releases](#).

Update using Microsoft Update

To receive IoT Edge for Linux on Windows updates, the Windows host should be configured to receive updates for other Microsoft products. By default, Microsoft Updates will be turned on during EFLOW installation. If custom configuration is needed after EFLOW installation, you can turn this option On/Off with the following steps:

1. Open **Settings** on the Windows host.
2. Select **Updates & Security**.
3. Select **Advanced options**.
4. Toggle the *Receive updates for other Microsoft products when you update Windows* button to **On**.

Update using Windows Server Update Services (WSUS)

On premises updates using WSUS is supported for IoT Edge for Linux on Windows updates. For more information about WSUS, see [Device Management Overview - WSUS](#).

Offline manual update

In some scenarios with restricted or limited internet connectivity, you may want to manually apply EFLOW updates offline. This is possible using Microsoft Update offline mechanisms. You can manually download and install an IoT Edge for Linux on Windows updates with the following steps:

1. Check the current EFLOW installed version. Open **Settings**, select **Apps -> Apps & features** search for *Azure IoT Edge*.
2. Search and download the required update from [EFLOW - Microsoft Update catalog](#).

3. Extract *AzureIoTEdge.msi* from the downloaded *.cab* file.

4. Install the extracted *AzureIoTEdge.msi*.

Managing Microsoft Updates

As explained before, IoT Edges for Linux on Windows updates are serviced using Microsoft Update channel, so turn on/off EFLOW updates, you'll have to manage Microsoft Updates. Listed below are some of the ways to automate turning on/off Microsoft updates. For more information about managing OS updates, see [OS Updates](#).

1. **CSP Policies** - By using the **Update/AllowMUUpdateService** CSP Policy - For more information about Microsoft Updates CSP policy, see [Policy CSP - MU Update](#).
2. **Manually manage Microsoft Updates** - For more information about how to Opt-In to Microsoft Updates, see [Opt-In to Microsoft Update](#).

Migration between EFLOW 1.1LTS and EFLOW 1.4LTS

IoT Edge for Linux on Windows doesn't support migrations between the different release trains. If you want to move from the 1.1LTS or 1.4LTS version to the Continuous Release (CR) version or viceversa, you'll have to uninstall the current version and install the new desired version.

Migration between EFLOW 1.1LTS to EFLOW 1.4LTS was introduced as part of EFLOW 1.1LTS ([1.1.2212.12122](#)) update. This migration will handle the EFLOW VM migration from 1.1LTS version to 1.4LTS version, including the following:

- IoT Edge runtime
- IoT Edge configurations
- Containers
- Networking and VM configuration
- Stored files

To migrate between EFLOW 1.1LTS to EFLOW 1.4LTS, use the following steps.

1. Get the latest Azure EFLOW 1.1LTS ([1.1.2212.12122](#)) update. If you're using Windows Update, *Check Updates* to get the latest EFLOW update.
2. For auto-download migration (needs Internet connection), skip this step. If the EFLOW VM has limited/no internet access, download the necessary files before

starting the migration.

- [1.4.2.12122 Standalone MSI ↗](#)
- [1.4.2.12122 Update MSI ↗](#)

3. Open an elevated PowerShell session

4. Start the EFLOW migration

 **Note**

You can migrate with one single cmdlet by using the `-autoConfirm` flag with the `Start-EflowMigration` cmdlet. If specified `Confirm-EflowMigration` doesn't need to be called to proceed with 1.4 migration.

a. If you're using the auto-download migration option run the following cmdlet

```
PowerShell
```

```
Start-EflowMigration
```

b. If you download the MSI on **Step 2**, use the downloaded files to apply the migration

```
PowerShell
```

```
Start-EflowMigration -standaloneMsiPath "<path-to-  
folder>\AzureIoTEdge LTS_1.4.2.12122_X64.msi"
```

5. Confirm the EFLOW migration

a. If you're using the auto-download migration option run the following cmdlet

```
PowerShell
```

```
Confirm-EflowMigration
```

b. If you download the MSI on **Step 2**, use the downloaded files to apply the migration

```
PowerShell
```

```
Confirm-EflowMigration -updateMsiPath "<path-to-  
folder>\AzureIoTEdge LTS_Update_1.4.2.12122_X64.msi"
```

 **Warning**

If for any reason the migration fails, the EFLOW VM will be restored to its original 1.1LTS version. If you want to cancel the migration or manually restore the EFLOW VM to prior state, you can use the following cmdlets `Start-EflowMigration` and then `Restore-EflowPriorToMigration`.

For more information, check `Start-EflowMigration`, `Confirm-EflowMigration` and `Restore-EflowPriorToMigration` cmdlet documentation by using the `Get-Help <cmdlet> -full` command.

Next steps

View the latest [IoT Edge for Linux on Windows releases](#).

Read more about [IoT Edge for Linux on Windows security premises](#).

IoT Edge for Linux on Windows security

Article • 06/04/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge for Linux on Windows benefits from all the security offerings from running on a Windows Client/Server host and ensures all the extra components keep the same security premises. This article provides information about the different security premises that are enabled by default, and some of the optional premises the user may enable.

Virtual machine security

The IoT Edge for Linux (EFLW) curated virtual machine is based on [Microsoft CBL-Mariner](#). CBL-Mariner is an internal Linux distribution for Microsoft's cloud infrastructure and edge products and services. CBL-Mariner is designed to provide a consistent platform for these devices and services and enhances Microsoft's ability to stay current on Linux updates. For more information, see [CBL-Mariner security](#).

The EFLW virtual machine is built on a four-point comprehensive security platform:

1. Servicing updates
2. Read-only root filesystem
3. Firewall lockdown
4. DM-Verity

Servicing updates

When security vulnerabilities arise, CBL-Mariner makes the latest security patches and fixes available for being serviced through ELOW monthly updates. The virtual machine has no package manager, so it's not possible to manually download and install RPM packages. All updates to the virtual machine are installed using ELOW A/B update mechanism. For more information on ELOW updates, see [Update IoT Edge for Linux on Windows](#)

Read-only root filesystem

The EFLOW virtual machine is made up of two main partitions *rootfs*, and *data*. The rootFS-A or rootFS-B partitions are interchangeable and one of the two is mounted as a read-only filesystem at `/`, which means that no changes are allowed on files stored inside this partition. On the other hand, the *data* partition mounted under `/var` is readable and writeable, allowing the user to modify the content inside the partition. The data stored on this partition isn't manipulated by the update process and hence won't be modified across updates.

Because you may need write access to `/etc`, `/home`, `/root`, `/var` for specific use cases, write access for these directories is done by overlaying them onto our data partition specifically to the directory `/var/.eflow/overlays`. The end result of this is that users can write anything to the previous mentioned directories. For more information about overlays, see [overlays ↗](#).



[\[+\] Expand table](#)

| Partition | Size | Description |
|-----------|--------------|---|
| BootEFIA | 8 MB | Firmware partition A for future GRUBless boot |
| BootA | 192 MB | Contains the bootloader for A partition |
| RootFS A | 4 GB | One of two active/passive partitions holding the root file system |
| BootEFIB | 8 MB | Firmware partition B for future GRUBless boot |
| BootB | 192 MB | Contains the bootloader for B partition |
| RootFS B | 4 GB | One of two active/passive partitions holding the root file system |
| Log | 1 GB or 6 GB | Logs specific partition mounted under <code>/logs</code> |
| Data | 2 GB to 2 TB | Stateful partition for storing persistent data across updates. Expandable according to the deployment configuration |

ⓘ Note

The partition layout represents the logical disk size and does not indicate the physical space the virtual machine will occupy on the host OS disk.

Firewall

By default, the EFLOW virtual machine uses [iptables](#) utility for firewall configurations. *Iptables* is used to set up, maintain, and inspect the tables of IP packet filter rules in the Linux kernel. The default implementation only allows incoming traffic on port 22 (SSH service) and blocks the traffic otherwise. You can check the *iptables* configuration with the following steps:

1. Open an elevated PowerShell session
2. Connect to the EFLOW virtual machine

```
PowerShell
```

```
Connect-EflowVm
```

3. List all the *iptables* rules

```
Bash
```

```
sudo iptables -L
```

```
iotedge-user@DESKTOP-FCABRER-EFLOW [ ~ ]$ sudo iptables -L
Chain INPUT (policy DROP)
target     prot opt source               destination
ACCEPT    all  --  anywhere             anywhere
ACCEPT    all  --  anywhere             anywhere             ctstate RELATED,ESTABLISHED
ACCEPT    tcp  --  anywhere             anywhere            tcp dpt:ssh

Chain FORWARD (policy DROP)
target     prot opt source               destination
DOCKER-USER all  --  anywhere             anywhere
DOCKER-ISOLATION-STAGE-1 all  --  anywhere             anywhere
ACCEPT    all  --  anywhere             anywhere           ctstate RELATED,ESTABLISHED
DOCKER    all  --  anywhere             anywhere
ACCEPT    all  --  anywhere             anywhere
ACCEPT    all  --  anywhere             anywhere

Chain OUTPUT (policy DROP)
target     prot opt source               destination
ACCEPT    all  --  anywhere             anywhere

Chain DOCKER (1 references)
target     prot opt source               destination

Chain DOCKER-ISOLATION-STAGE-1 (1 references)
target     prot opt source               destination
DOCKER-ISOLATION-STAGE-2 all  --  anywhere             anywhere
RETURN    all  --  anywhere             anywhere

Chain DOCKER-ISOLATION-STAGE-2 (1 references)
target     prot opt source               destination
DROP      all  --  anywhere             anywhere
RETURN    all  --  anywhere             anywhere

Chain DOCKER-USER (1 references)
target     prot opt source               destination
RETURN    all  --  anywhere             anywhere
```

Verified boot

The EFLOW virtual machine supports **Verified boot** through the included *device-mapper-verity* (*dm-verity*) kernel feature, which provides transparent integrity checking of block devices. *dm-verity* helps prevent persistent rootkits that can hold onto root privileges and compromise devices. This feature assures the virtual machine base software image it's the same and it wasn't altered. The virtual machine uses the *dm-verity* feature to check specific block device, the underlying storage layer of the file system, and determine if it matches its expected configuration.

By default, this feature is disabled in the virtual machine, and can be turned on or off. For more information, see [dm-verity](#).

Trusted platform module (TPM)

[Trusted platform module \(TPM\)](#) technology is designed to provide hardware-based, security-related functions. A TPM chip is a secure crypto-processor that is designed to carry out cryptographic operations. The chip includes multiple physical security mechanisms to make it tamper resistant, and malicious software is unable to tamper with the security functions of the TPM.

The EFLOW virtual machine doesn't support vTPM. However, the user can enable/disable the TPM passthrough feature that allows the EFLOW virtual machine to use the Windows host OS TPM. This enables two main scenarios:

- Use TPM technology for IoT Edge device provisioning using Device Provision Service (DPS). For more information, see [Create and provision an IoT Edge for Linux on Windows device at scale by using a TPM](#).
- Read-only access to cryptographic keys stored inside the TPM. For more information, see [Set-EflowVmFeature to enable TPM passthrough](#).

Secure host & virtual machine communication

EFLOW provides multiple ways to interact with the virtual machine by exposing a rich PowerShell module implementation. For more information, see [PowerShell functions for IoT Edge for Linux on Windows](#). This module requires an elevated session to run, and it's signed using a Microsoft Corporation certificate.

All communications between the Windows host operating system and the EFLOW virtual machine required by the PowerShell cmdlets are done using an SSH channel. By default, the virtual machine SSH service won't allow authentication via username and password,

and it's limited to certificate authentication. The certificate is created during EFLOW deployment process, and is unique for each EFLOW installation. Furthermore, to prevent SSH brute force attacks, the virtual machine blocks an IP address if it attempts more than three connections per minute to SSH service.

In the EFLOW Continuous Release (CR) version, we introduced a change in the transport channel used to establish the SSH connection. Originally, SSH service runs on TCP port 22, which can be accessed by all external devices in the same network using a TCP socket to that specific port. For security reasons, EFLOW CR runs the SSH service over Hyper-V sockets instead of normal TCP sockets. All communication over Hyper-V sockets runs between the Windows host OS and the EFLOW virtual machine, without using networking. This limits the access of the SSH service, restricting connections to only the Windows host OS. For more information, see [Hyper-V sockets](#).

Next steps

Read more about [Windows IoT security premises](#)

Stay up-to-date with the latest [IoT Edge for Linux on Windows updates](#).

IoT Edge for Linux on Windows networking

Article • 06/04/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides information about how to configure the networking between the Windows host OS and the IoT Edge for Linux on Windows (EFLOW) virtual machine. EFLOW uses a [CBL-Mariner](#) ↗ Linux virtual machine in order to run IoT Edge modules. For more information about EFLOW architecture, see [What is Azure IoT Edge for Linux on Windows](#).

Networking

To establish a communication channel between the Windows host OS and the EFLOW virtual machine, we use Hyper-V networking stack. For more information about Hyper-V networking, see [Hyper-V networking basics](#). Basic networking in EFLOW is simple; it uses two parts, a virtual switch and a virtual network.

The easiest way to establish basic networking on Windows client SKUs is by using the [default switch](#) ↗ already created by the Hyper-V feature. During EFLOW deployment, if no specific virtual switch is specified using the `-vSwitchName` and `-vSwitchType` flags, the virtual machine is created using the **default switch**.

On Windows Server SKUs devices, networking is a bit more complicated as there's no **default switch** available. However, there's a comprehensive guide on [Azure IoT Edge for Linux on Windows virtual switch creation](#).

To handle different types of networking, you can use different types of virtual switches and add multiple virtual network adapters.

Virtual switch choices

EFLOW supports two types of Hyper-V virtual switches: **internal** and **external**. You'll choose which one of each you want when you create it previous to EFLOW deployment. You can use Hyper-V Manager or the Hyper-V module for Windows PowerShell to create and manage virtual switches. For more information about creating a virtual switch, see [Create a virtual switch for Hyper-V virtual machines](#).

You can make some changes to a virtual switch after you create it. For example, it's possible to change an existing switch to a different type, but doing that may affect the networking capabilities of EFLOW virtual machine connected to that switch. So, it's not recommended to do change the virtual switch configuration unless you made a mistake or need to test something.

Depending if the EFLOW VM is deployed in a Windows client SKU or Windows Server SKU device, we support different types of switches, as shown in the following table.

[] Expand table

| Virtual switch type | Windows client SKUs | Windows Server SKUs |
|---------------------|---------------------|---------------------|
| External | ✓ | ✓ |
| Internal | - | ✓ |
| Default switch | ✓ | - |

- **External virtual switch** - Connects to a wired, physical network by binding to a physical network adapter. It gives virtual machines access to a physical network to communicate with devices on an external network. In addition, it allows virtual machines on the same Hyper-V server to communicate with each other.
- **Internal virtual switch** - Connects to a network that can be used only by the virtual machines running on the host that has the virtual switch, and between the host and the virtual machines.

! Note

The **default switch** is a particular internal virtual switch created by default once Hyper-V is enabled in Windows client SKUs. The virtual switch already has a DHCP Server for IP assignments, Internet Connection Sharing (ICS) enabled, and a NAT table. For EFLOW purposes, the Default Switch is a Virtual Internal Switch that can be used without further configuration.

IP address allocations

To enable EFLOW VM network IP communications, the virtual machine must have an IP address assigned. This IP address can be configured by two different methods: **Static IP** or **DHCP**.

Depending on the type of virtual switch used, EFLOW VM supports different IP allocations, as shown in the following table.

[+] Expand table

| Virtual switch type | Static IP | DHCP |
|---------------------|-----------|------|
| External | ✓ | ✓ |
| Internal | ✓ | ✓ |
| Default switch | - | ✓ |

- **Static IP** - This IP address is permanently assigned to the EFLOW VM during installation and doesn't change across EFLOW VM or Windows host reboots. Static IP addresses typically have two versions: IPv4 and IPv6; however, EFLOW only supports static IP for IPv4 addresses. On networks using static IP, each device on the network has its address with no overlap. During EFLOW installation, you must input the **EFLOW VM IP4 address**(`-ip4Address`), the **IP4 prefix length**(`-ip4PrefixLength`), and the **default gateway IP4 address**(`-ip4GatewayAddress`). All three parameters must be input for correct configuration.

For example, if you want to deploy the EFLOW VM using an *external virtual switch* named *ExternalEflow* with a static IP address *192.168.0.100*, default gateway *192.168.0.1*, and a prefix length of 24, the following deploy command is needed

PowerShell

```
Deploy-Eflow -vSwitchName "ExternalEflow" -vswitchType "External" -  
ip4Address 192.168.0.100 -ip4GatewayAddress 192.168.0.1 -  
ip4PrefixLength 24
```

⚠ Warning

When using static IP, the **three parameters** (`ip4Address`, `ip4GatewayAddress`, `ip4PrefixLength`) must be used. Also, if the IP address is invalid, being used by another device on the network, or the gateway address is incorrect, EFLOW installation could fail as the EFLOW VM can't get an IP address.

- **DHCP** - Contrary to static IP, when using DHCP, the EFLOW virtual machine is assigned with a dynamic IP address; which is an address that may change. The network must have a DHCP server configured and operating to assign dynamic IP addresses. The DHCP server assigns a vacant IP address to the EFLOW VM and others connected to the network. Therefore, when deploying EFLOW using DHCP, no IP address, gateway address, or prefix length is needed, as the DHCP server provides all the information.

Warning

When deploying EFLOW using DHCP, a DHCP server must be present in the network connected to the EFLOW VM virtual switch. If no DHCP server is present, EFLOW installation will fail as the VM can't get an IP address.

DNS

Domain Name System (DNS) translates human-readable domain names (for example, www.microsoft.com) to machine-readable IP addresses (for example, 192.0.2.44). The EFLOW virtual machine uses [systemd](#) (system and service manager), so the DNS or name resolution services are provided to local applications and services via the [systemd-resolved](#) service.

By default, the EFLOW VM DNS configuration file contains the local stub 127.0.0.53 as the only DNS server. This is redirected to the `/etc/resolv.conf` file, which is used to add the name servers used by the system. The local stub is a DNS server that runs locally to resolve DNS queries. In some cases, these queries are forwarded to another DNS server in the network and then cached locally.

It's possible to configure the EFLOW virtual machine to use a specific DNS server, or list of servers. To do so, you can use the `Set-EflowVmDnsServers` PowerShell cmdlet. For more information about DNS configuration, see [PowerShell functions for IoT Edge for Linux on Windows](#).

To check the DNS servers assigned to the EFLOW VM, from inside the EFLOW VM, use the command: `resolvectl status`. The command's output shows a list of the DNS servers configured for each interface. In particular, it's important to check the `eth0` interface status, which is the default interface for the EFLOW VM communication. Also, make sure to check the IP addresses of the **Current DNS Servers** and **DNS Servers** fields of the list. If there's no IP address, or the IP address isn't a valid DNS server IP address, then the DNS service won't work.

```
iotedge-user@LAPTOP-80QC0NUB-EFLOW [ ~ ]$ resolvectl
Global
    LLMNR setting: no
MulticastDNS setting: yes
    DNSOverTLS setting: no
        DNSSEC setting: no
        DNSSEC supported: no
Fallback DNS Servers: 8.8.8.8
                        8.8.4.4
                        2001:4860:4860::8888
                        2001:4860:4860::8844
        DNSSEC NTA: 10.in-addr.arpa
                    16.172.in-addr.arpa
                    168.192.in-addr.arpa
                    17.172.in-addr.arpa
                    18.172.in-addr.arpa
                    19.172.in-addr.arpa
                    20.172.in-addr.arpa
                    21.172.in-addr.arpa
                    22.172.in-addr.arpa
                    23.172.in-addr.arpa
                    24.172.in-addr.arpa
                    25.172.in-addr.arpa
                    26.172.in-addr.arpa
                    27.172.in-addr.arpa
                    28.172.in-addr.arpa
                    29.172.in-addr.arpa
                    30.172.in-addr.arpa
                    31.172.in-addr.arpa
                    corp
                    d.f.ip6.arpa
                    home
                    internal
                    intranet
                    lan
                    local
                    private
                    test

Link 3 (docker0)
    Current Scopes: none
        LLMNR setting: yes
MulticastDNS setting: no
    DNSOverTLS setting: no
        DNSSEC setting: no
        DNSSEC supported: no

Link 2 (eth0)
    Current Scopes: DNS
        LLMNR setting: yes
MulticastDNS setting: no
    DNSOverTLS setting: no
        DNSSEC setting: no
        DNSSEC supported: no
    Current DNS Server: 172.18.224.1
    DNS Servers: 172.18.224.1
```

Static MAC Address

Hyper-V allows you to create virtual machines with a **static** or **dynamic** MAC address. During EFLOW virtual machine creation, the MAC address is randomly generated and stored locally to keep the same MAC address across virtual machine or Windows host

reboots. To query the EFLOW virtual machine MAC address, you can use the following command.

```
PowerShell
```

```
Get-EflowVmAddr
```

Multiple Network Interface Cards (NICs)

There are many network virtual appliances and scenarios that require multiple NICs. The EFLOW virtual machine supports attaching multiple NICs. With multiple NICs, you can better manage your network traffic. You can also isolate traffic between the frontend NIC and backend NICs, or separating data plane traffic from the management plane communication.

For example, there are numerous of industrial IoT scenarios that require connecting the EFLOW virtual machine to a demilitarized zone (DMZ), and to the offline network where all the OPC UA compliant devices are connected. This is just one of the multiple scenarios that can be supported by attaching multiple NICs to the EFLOW VM.

For more information about multiple NICs, see [Multiple NICs support ↗](#).

⚠ Warning

When using EFLOW multiple NICs feature, you may want to set up the different routes priorities. By default, EFLOW will create one default route per *ehtX* interface assigned to the VM and assign a random priority. If all interfaces are connected to the internet, random priorities may not be a problem. However, if one of the NICs is connected to an offline network, you may want to prioritize the online NIC over the offline NIC to get the EFLOW VM connected to the internet. For more information about custom routing, see [EFLOW routing ↗](#).

Next steps

Read more about [Azure IoT Edge for Linux on Windows Security](#).

Learn how to manage EFLOW networking [Networking configuration for Azure IoT Edge for Linux on Windows](#)

Azure IoT Edge for Linux on Windows supported systems

Article • 06/05/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides details about which systems support IoT Edge for Linux on Windows, whether generally available or in preview.

Get support

If you experience problems while using Azure IoT Edge for Linux on Windows, there are several ways to seek support. Try one of the following channels for support:

Reporting bugs - Bugs can be reported on the [issues page](#) of the project. Bugs related to Azure IoT Edge can be reported on the [IoT Edge issues page](#). Fixes rapidly make their way from the projects in to product updates.

Microsoft Customer Support team - Users who have a [support plan](#) can engage the Microsoft Customer Support team by creating a support ticket directly from the [Azure portal](#).

Container engines

By default, Azure IoT Edge for Linux on Windows includes IoT Edge runtime as part of the virtual machine composition. The IoT Edge runtime provides moby-engine as the container engine, to run modules implemented as containers. This container engine is based on the Moby open-source project. For more information about container engines, support, and IoT Edge, see [IoT Edge Platform support](#).

Operating systems

IoT Edge for Linux on Windows uses IoT Edge in a Linux virtual machine running on a Windows host. In this way, you can run Linux modules on a Windows device. Azure IoT Edge for Linux on Windows runs on the following Windows SKUs:

- **Windows Client**
 - Pro, Enterprise, IoT Enterprise SKUs
 - Windows 10 - Minimum build 17763 with all current cumulative updates installed
 - Windows 11
- **Windows Server**
 - Windows Server 2019 - Minimum build 17763 with all current cumulative updates installed
 - Windows Server 2022

Platform support

Azure IoT Edge for Linux on Windows supports the following architectures:

[\[+\] Expand table](#)

| Version | AMD64 | ARM64 |
|--|-------|-------|
| EFLOW 1.1 LTS | | |
| EFLOW Continuous Release (CR) (Public preview ↗) | | |
| EFLOW 1.4 LTS | | |

For more information about Windows ARM64 supported processors, see [Windows Processor Requirements](#).

Nested virtualization

Azure IoT Edge for Linux on Windows (EFLW) can run in Windows virtual machines. Using a virtual machine as an IoT Edge device is common when customers want to augment existing infrastructure with edge intelligence. In order to run the EFLW virtual machine inside a Windows VM, the host VM must support nested virtualization. EFLW supports the following nested virtualization scenarios:

[\[+\] Expand table](#)

| Version | Hyper-V
VM | Azure
VM | VMware ESXi
VM | Other
Hypervisor |
|--|---------------|-------------|-------------------|---------------------|
| EFLOW 1.1 LTS | ✓ | ✓ | ✓ | - |
| EFLOW Continuous Release (CR)
(Public preview ) | ✓ | ✓ | ✓ | - |
| EFLOW 1.4 LTS | ✓ | ✓ | ✓ | - |

For more information, see [EFLow Nested virtualization](#).

VMware virtual machine

Azure IoT Edge for Linux on Windows supports running inside a Windows virtual machine running on top of [VMware ESXi !\[\]\(420f3223eaa712aeb8cc2846ee8c7fc1_img.jpg\)](#) product family. Specific networking and virtualization configurations are needed to support this scenario. For more information about VMware configuration, see [EFLow Nested virtualization](#).

Releases

IoT Edge for Linux on Windows release assets and release notes are available on the [iotedge-eflow releases !\[\]\(534435ed307b6f2be95828c9cecf217a_img.jpg\)](#) page. This section reflects information from those release notes to help you visualize the components of each version more easily.

The following table lists the components included in each release. Each release train is independent, and we don't guarantee backwards compatibility and migration between versions. For more information about IoT Edge version, see [IoT Edge platform support](#).

 Expand table

| Release | IoT Edge | CBL-Mariner | Defender for IoT |
|--------------------|----------|-------------|------------------|
| 1.1 LTS | 1.1 | 2.0 | - |
| Continuous Release | 1.3 | 2.0 | 3.12.3 |
| 1.4 LTS | 1.4 | 2.0 | 3.12.3 |

Minimum system requirements

Azure IoT Edge for Linux on Windows runs great on small edge devices to server grade hardware. Choosing the right hardware for your scenario depends on the workloads that

you want to run.

A Windows device with the following minimum requirements:

- Hardware requirements
 - Minimum Free Memory: 1 GB
 - Minimum Free Disk Space: 10 GB
- Virtualization support
 - On Windows 10, enable Hyper-V. For more information, see [Install Hyper-V on Windows 10](#).
 - On Windows Server, install the Hyper-V role and create a default network switch. For more information, see [Nested virtualization for Azure IoT Edge for Linux on Windows](#).
 - On a virtual machine, configure nested virtualization. For more information, see [nested virtualization](#).

Next steps

Read more about [IoT Edge for Linux on Windows security premises](#).

Stay up-to-date with the latest [IoT Edge for Linux on Windows updates](#).

Create and provision an IoT Edge for Linux on Windows device using X.509 certificates

Article • 06/03/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides end-to-end instructions for registering and provisioning an IoT Edge for Linux on Windows device.

Every device that connects to an IoT hub has a device ID that's used to track cloud-to-device or device-to-cloud communications. You configure a device with its connection information, which includes the IoT hub hostname, the device ID, and the information the device uses to authenticate to IoT Hub.

The steps in this article walk through a process called manual provisioning, where you connect a single device to its IoT hub. For manual provisioning, you have two options for authenticating IoT Edge devices:

- **Symmetric keys:** When you create a new device identity in IoT Hub, the service creates two keys. You place one of the keys on the device, and it presents the key to IoT Hub when authenticating.

This authentication method is faster to get started, but not as secure.

- **X.509 self-signed:** You create two X.509 identity certificates and place them on the device. When you create a new device identity in IoT Hub, you provide thumbprints from both certificates. When the device authenticates to IoT Hub, it presents one certificate and IoT Hub verifies that the certificate matches its thumbprint.

This authentication method is more secure and recommended for production scenarios.

This article covers using X.509 certificates as your authentication method. If you want to use symmetric keys, see [Create and provision an IoT Edge for Linux on Windows device](#)

using symmetric keys.

ⓘ Note

If you have many devices to set up and don't want to manually provision each one, use one of the following articles to learn how IoT Edge works with the IoT Hub device provisioning service:

- [Create and provision IoT Edge devices at scale using X.509 certificates](#)
- [Create and provision IoT Edge devices at scale with a TPM](#)
- [Create and provision IoT Edge devices at scale using symmetric keys](#)

Prerequisites

This article covers registering your IoT Edge device and installing IoT Edge for Linux on Windows. These tasks have different prerequisites and utilities used to accomplish them. Make sure you have all the prerequisites covered before proceeding.

Device management tools

You can use the **Azure portal**, **Visual Studio Code**, or **Azure CLI** for the steps to register your device. Each utility has its own prerequisites or may need to be installed:

Portal

A free or standard [IoT hub](#) in your Azure subscription.

Device requirements

A Windows device with the following minimum requirements:

- System Requirements
 - Windows 10¹/11 (Pro, Enterprise, IoT Enterprise)
 - Windows Server 2019¹/2022

¹ Windows 10 and Windows Server 2019 minimum build 17763 with all current cumulative updates installed.

- Hardware requirements
 - Minimum Free Memory: 1 GB
 - Minimum Free Disk Space: 10 GB

- Virtualization support
 - On Windows 10, enable Hyper-V. For more information, see [Install Hyper-V on Windows 10](#).
 - On Windows Server, install the Hyper-V role and create a default network switch. For more information, see [Nested virtualization for Azure IoT Edge for Linux on Windows](#).
 - On a virtual machine, configure nested virtualization. For more information, see [nested virtualization](#).
- Networking support
 - Windows Server does not come with a default switch. Before you can deploy EFLOW to a Windows Server device, you need to create a virtual switch. For more information, see [Create virtual switch for Linux on Windows](#).
 - Windows Desktop versions come with a default switch that can be used for EFLOW installation. If needed, you can create your own custom virtual switch.

💡 Tip

If you want to use **GPU-accelerated Linux modules** in your Azure IoT Edge for Linux on Windows deployment, there are several configuration options to consider.

You will need to install the correct drivers depending on your GPU architecture, and you may need access to a Windows Insider Program build. To determine your configuration needs and satisfy these prerequisites, see [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

Make sure you take the time to satisfy the prerequisites for GPU acceleration now. You will need to restart the installation process if you decide you want GPU acceleration during installation.

Developer tools

Prepare your target device for the installation of Azure IoT Edge for Linux on Windows and the deployment of the Linux virtual machine:

1. Set the execution policy on the target device to `AllSigned`. You can check the current execution policy in an elevated PowerShell prompt using the following command:

```
PowerShell
```

```
Get-ExecutionPolicy -List
```

If the execution policy of `local machine` is not `AllSigned`, you can set the execution policy using:

PowerShell

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Force
```

For more information on the Azure IoT Edge for Linux on Windows PowerShell module, see the [PowerShell functions reference](#).

Generate device identity certificates

Manual provisioning with X.509 certificates requires IoT Edge version 1.0.10 or newer.

When you provision an IoT Edge device with X.509 certificates, you use what's called a *device identity certificate*. This certificate is only used for provisioning an IoT Edge device and authenticating the device with Azure IoT Hub. It's a leaf certificate that doesn't sign other certificates. The device identity certificate is separate from the certificate authority (CA) certificates that the IoT Edge device presents to modules or downstream devices for verification.

For X.509 certificate authentication, each device's authentication information is provided in the form of *thumbprints* taken from your device identity certificates. These thumbprints are given to IoT Hub at the time of device registration so that the service can recognize the device when it connects.

For more information about how the CA certificates are used in IoT Edge devices, see [Understand how Azure IoT Edge uses certificates](#).

You need the following files for manual provisioning with X.509:

- Two device identity certificates with their matching private key certificates in .cer or .pem formats. You need two device identity certificates for certificate rotation. A best practice is to prepare two different device identity certificates with different expiration dates. If one certificate expires, the other is still valid and gives you time to rotate the expired certificate.

One set of certificate and key files is provided to the IoT Edge runtime. When you create device identity certificates, set the certificate common name (CN) with the device ID that you want the device to have in your IoT hub.

- Thumbprints taken from both device identity certificates. IoT Hub requires two thumbprints when registering an IoT Edge device. You can use only one certificate for registration. To use a single certificate, set the same certificate thumbprint for both the primary and secondary thumbprints when registering the device.

Thumbprint values are 40-hex characters for SHA-1 hashes or 64-hex characters for SHA-256 hashes. Both thumbprints are provided to IoT Hub at the time of device registration.

One way to retrieve the thumbprint from a certificate is with the following openssl command:

```
Windows Command Prompt
```

```
openssl x509 -in <certificate filename>.pem -text -fingerprint
```

The thumbprint is included in the output of this command. For example:

```
Windows Command Prompt
```

```
SHA1  
Fingerprint=D2:68:D9:04:9F:1A:4D:6A:FD:84:77:68:7B:C6:33:C0:32:37:51:12
```

If you don't have certificates available, you can [Create demo certificates to test IoT Edge device features](#). Follow the instructions in that article to set up certificate creation scripts, create a root CA certificate, and create a IoT Edge device identity certificate. For testing, you can create a single device identity certificate and use the same thumbprint for both primary and secondary thumbprint values when registering the device in IoT Hub.

Register your device

You can use the [Azure portal](#), [Visual Studio Code](#), or [Azure CLI](#) to register your device, depending on your preference.

Portal

In your IoT hub in the Azure portal, IoT Edge devices are created and managed separately from IoT devices that aren't edge enabled.

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. In the left pane, select **Devices** from the menu, then select **Add Device**.

3. On the **Create a device** page, provide the following information:

- Create a descriptive device ID. Make a note of this device ID, as you use it later.
- Check the **IoT Edge Device** checkbox.
- Select **X.509 Self-Signed** as the authentication type.
- Provide the primary and secondary identity certificate thumbprints.

Thumbprint values are 40-hex characters for SHA-1 hashes or 64-hex characters for SHA-256 hashes. The Azure portal supports hexadecimal values only. Remove column separators and spaces from the thumbprint values before entering them in the portal. For example,

D2:68:D9:04:9F:1A:4D:6A:FD:84:77:68:7B:C6:33:C0:32:37:51:12 is entered as D268D9049F1A4D6AFD8477687BC633C032375112.

Tip

If you are testing and want to use one certificate, you can use the same certificate for both the primary and secondary thumbprints.

4. Select **Save**.

Now that you have a device registered in IoT Hub, retrieve the information that you use to complete installation and provisioning of the IoT Edge runtime.

View registered devices and retrieve provisioning information

Devices that use X.509 certificate authentication need their IoT hub name, their device name, and their certificate files to complete installation and provisioning of the IoT Edge runtime.

Portal

The edge-enabled devices that connect to your IoT hub are listed on the **Devices** page. You can filter the list by device type *IoT Edge devices*.

Install IoT Edge

Deploy Azure IoT Edge for Linux on Windows on your target device.

ⓘ Note

The following PowerShell process outlines how to deploy IoT Edge for Linux on Windows onto the local device. To deploy to a remote target device using PowerShell, you can use [Remote PowerShell](#) to establish a connection to a remote device and run these commands remotely on that device.

1. In an elevated PowerShell session, run either of the following commands depending on your target device architecture to download IoT Edge for Linux on Windows.

- X64/AMD64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_X64" -OutFile
$msiPath
```

- ARM64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_ARM64" -
OutFile $msiPath
```

2. Install IoT Edge for Linux on Windows on your device.

```
PowerShell

Start-Process -Wait msieexec -ArgumentList
"/i","$([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))","/qn"
```

You can specify custom IoT Edge for Linux on Windows installation and VHDX directories by adding `INSTALLDIR=<FULLY_QUALIFIED_PATH>` and `VHDXDIR=<FULLY_QUALIFIED_PATH>` parameters to the install command. For example, if you want to use the *D:\EFLOW* folder for installation and the *D:\EFLOW-VHDX* for the VHDX, you can use the following PowerShell cmdlet.

PowerShell

```
Start-Process -Wait msieexec -ArgumentList  
"/i","$([io.Path]::Combine($env:TEMP,  
'AzureIoTEdge.msi'))","/qn","INSTALLDIR=D:\EFLOW", "VHDXDIR=D:\EFLOW-  
VHDX"
```

3. Set the execution policy on the target device to `AllSigned` if it is not already. See the PowerShell prerequisites for commands to check the current execution policy and set the execution policy to `AllSigned`.
4. Create the IoT Edge for Linux on Windows deployment. The deployment creates your Linux virtual machine and installs the IoT Edge runtime for you.

PowerShell

`Deploy-Eflow`

💡 Tip

By default, the `Deploy-Eflow` command creates your Linux virtual machine with 1 GB of RAM, 1 vCPU core, and 16 GB of disk space. However, the resources your VM needs are highly dependent on the workloads you deploy. If your VM does not have sufficient memory to support your workloads, it will fail to start.

You can customize the virtual machine's available resources using the `Deploy-Eflow` command's optional parameters. This is required to deploy EFLOW on a device with the minimum hardware requirements.

For example, the command below creates a virtual machine with 1 vCPU core, 1 GB of RAM (represented in MB), and 2 GB of disk space:

PowerShell

```
Deploy-Eflow -cpuCount 1 -memoryInMB 1024 -vmDataSize 2
```

For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

By default, the EFLOW Linux virtual machine has no DNS configuration. Deployments using DHCP will try to obtain the DNS configuration propagated by the DHCP server. Please check your DNS configuration to ensure internet connectivity. For more information, see [AzEFLOW-DNS](#).

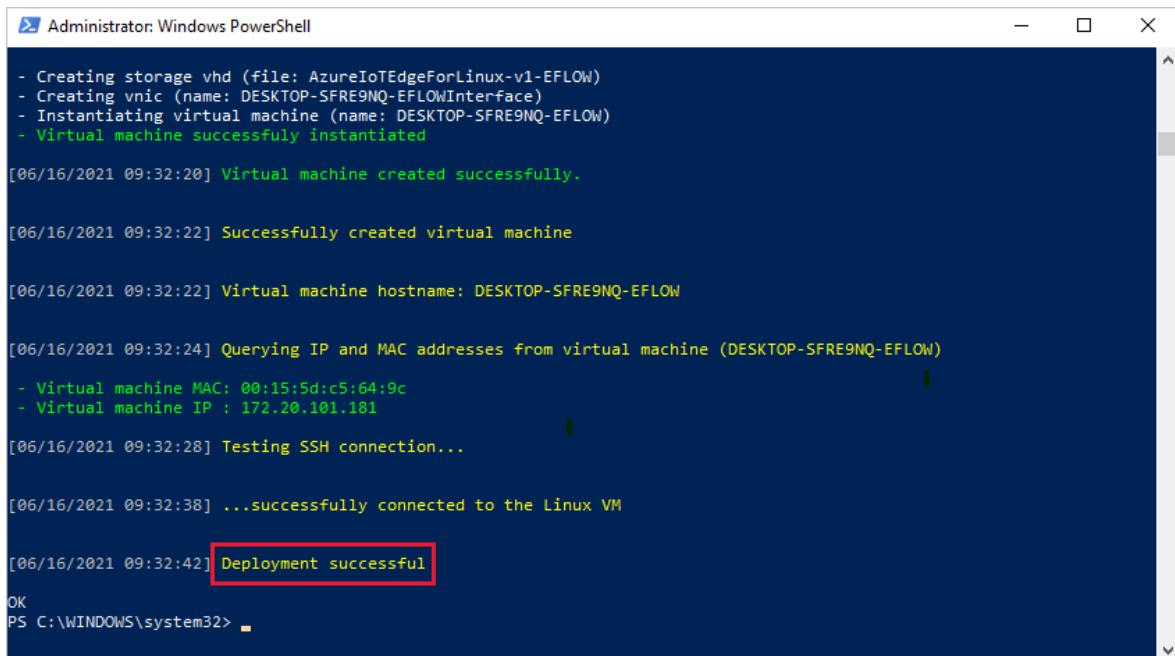
You can assign a GPU to your deployment to enable GPU-accelerated Linux modules. To gain access to these features, you will need to install the prerequisites detailed in [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

To use a GPU passthrough, add the `gpuName`, `gpuPassthroughType`, and `gpuCount` parameters to your `Deploy-Eflow` command. For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

Warning

Enabling hardware device passthrough may increase security risks. Microsoft recommends a device mitigation driver from your GPU's vendor, when applicable. For more information, see [Deploy graphics devices using discrete device assignment](#).

5. Enter 'Y' to accept the license terms.
6. Enter 'O' or 'R' to toggle **Optional diagnostic data** on or off, depending on your preference.
7. Once the deployment is complete, the PowerShell window reports **Deployment successful**.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The console output details the deployment process:

- Creating storage vhd (file: AzureIoTEdgeForLinux-v1-EFLOW)
- Creating vnic (name: DESKTOP-SFRE9NQ-EFLOWInterface)
- Instantiating virtual machine (name: DESKTOP-SFRE9NQ-EFLOW)
- Virtual machine successfully instantiated

[06/16/2021 09:32:20] Virtual machine created successfully.

[06/16/2021 09:32:22] Successfully created virtual machine

[06/16/2021 09:32:22] Virtual machine hostname: DESKTOP-SFRE9NQ-EFLOW

[06/16/2021 09:32:24] Querying IP and MAC addresses from virtual machine (DESKTOP-SFRE9NQ-EFLOW)

- Virtual machine MAC: 00:15:5d:c5:64:9c
- Virtual machine IP : 172.20.101.181

[06/16/2021 09:32:28] Testing SSH connection...

[06/16/2021 09:32:38] ...successfully connected to the Linux VM

[06/16/2021 09:32:42] Deployment successful

OK
PS C:\WINDOWS\system32> ■

After a successful deployment, you are ready to provision your device.

Provision the device with its cloud identity

You're ready to set up your device with its cloud identity and authentication information.

To provision your device using X.509 certificates, you need your **IoT hub name**, **device ID**, and the absolute paths to your **identity certificate** and **private key** on your Windows host machine.

Have the device identity certificate and its matching private key ready on your target device. Know the absolute path to both files.

Run the following command in an elevated PowerShell session on your target device. Replace the placeholder text with your own values.



PowerShell

```
Provision-EflowVm -provisioningType ManualX509 -iotHubHostname "HUB_HOSTNAME_HERE" -deviceId "DEVICE_ID_HERE" -identityCertPath "ABSOLUTE_PATH_TO_IDENTITY_CERT_HERE" -identityPrivKeyPath "ABSOLUTE_PATH_TO_PRIVATE_KEY_HERE"
```

For more information about the `Provision-EflowVM` command, see [PowerShell functions for IoT Edge for Linux on Windows](#).

Verify successful configuration

Verify that IoT Edge for Linux on Windows was successfully installed and configured on your IoT Edge device.

1. Sign in to your IoT Edge for Linux on Windows virtual machine using the following command in your PowerShell session:

```
PowerShell
```

```
Connect-EflowVm
```

 **Note**

The only account allowed to SSH to the virtual machine is the user that created it.

2. Once you are logged in, you can check the list of running IoT Edge modules using the following Linux command:

```
Bash
```

```
sudo iotedge list
```

3. If you need to troubleshoot the IoT Edge service, use the following Linux commands.

- a. Retrieve the service logs.

```
Bash
```

```
sudo iotedge system logs
```

- b. Use the `check` tool to verify configuration and connection status of the device.

```
Bash
```

```
sudo iotedge check
```

 **Note**

On a newly provisioned device, you may see an error related to IoT Edge Hub:

- ✗ production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error

Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module isn't running. To resolve the error, in IoT Hub, set the modules for the device and create a deployment. Creating a deployment for the device starts the modules on the device including the IoT Edge Hub module.

When you create a new IoT Edge device, it displays the status code `417 -- The device's deployment configuration is not set` in the Azure portal. This status is normal, and means that the device is ready to receive a module deployment.

Uninstall IoT Edge for Linux on Windows

If you want to remove the Azure IoT Edge for Linux on Windows installation from your device, use the following commands.

1. Open Settings on Windows
2. Select Add or Remove Programs
3. Select *Azure IoT Edge* app
4. Select Uninstall

Next steps

- Continue to [deploy IoT Edge modules](#) to learn how to deploy modules onto your device.
- Learn how to [manage certificates on your IoT Edge for Linux on Windows virtual machine](#) and transfer files from the host OS to your Linux virtual machine.
- Learn how to [configure your IoT Edge devices to communicate through a proxy server](#).

Create and provision an IoT Edge for Linux on Windows device using symmetric keys

Article • 06/03/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides end-to-end instructions for registering and provisioning an IoT Edge for Linux on Windows device.

Every device that connects to an IoT hub has a device ID that's used to track cloud-to-device or device-to-cloud communications. You configure a device with its connection information, which includes the IoT hub hostname, the device ID, and the information the device uses to authenticate to IoT Hub.

The steps in this article walk through a process called manual provisioning, where you connect a single device to its IoT hub. For manual provisioning, you have two options for authenticating IoT Edge devices:

- **Symmetric keys:** When you create a new device identity in IoT Hub, the service creates two keys. You place one of the keys on the device, and it presents the key to IoT Hub when authenticating.

This authentication method is faster to get started, but not as secure.

- **X.509 self-signed:** You create two X.509 identity certificates and place them on the device. When you create a new device identity in IoT Hub, you provide thumbprints from both certificates. When the device authenticates to IoT Hub, it presents one certificate and IoT Hub verifies that the certificate matches its thumbprint.

This authentication method is more secure and recommended for production scenarios.

This article covers using symmetric keys as your authentication method. If you want to use X.509 certificates, see [Create and provision an IoT Edge for Linux on Windows](#)

device using X.509 certificates.

Note

If you have many devices to set up and don't want to manually provision each one, use one of the following articles to learn how IoT Edge works with the IoT Hub device provisioning service:

- [Create and provision IoT Edge devices at scale using X.509 certificates](#)
- [Create and provision IoT Edge devices at scale with a TPM](#)
- [Create and provision IoT Edge devices at scale using symmetric keys](#)

Prerequisites

This article covers registering your IoT Edge device and installing IoT Edge for Linux on Windows. These tasks have different prerequisites and utilities used to accomplish them. Make sure you have all the prerequisites covered before proceeding.

Device management tools

You can use the **Azure portal**, **Visual Studio Code**, or **Azure CLI** for the steps to register your device. Each utility has its own prerequisites or may need to be installed:

Portal

A free or standard [IoT hub](#) in your Azure subscription.

Device requirements

A Windows device with the following minimum requirements:

- System Requirements
 - Windows 10¹/11 (Pro, Enterprise, IoT Enterprise)
 - Windows Server 2019¹/2022

¹ Windows 10 and Windows Server 2019 minimum build 17763 with all current cumulative updates installed.

- Hardware requirements
 - Minimum Free Memory: 1 GB
 - Minimum Free Disk Space: 10 GB

- Virtualization support
 - On Windows 10, enable Hyper-V. For more information, see [Install Hyper-V on Windows 10](#).
 - On Windows Server, install the Hyper-V role and create a default network switch. For more information, see [Nested virtualization for Azure IoT Edge for Linux on Windows](#).
 - On a virtual machine, configure nested virtualization. For more information, see [nested virtualization](#).
- Networking support
 - Windows Server does not come with a default switch. Before you can deploy EFLOW to a Windows Server device, you need to create a virtual switch. For more information, see [Create virtual switch for Linux on Windows](#).
 - Windows Desktop versions come with a default switch that can be used for EFLOW installation. If needed, you can create your own custom virtual switch.

💡 Tip

If you want to use **GPU-accelerated Linux modules** in your Azure IoT Edge for Linux on Windows deployment, there are several configuration options to consider.

You will need to install the correct drivers depending on your GPU architecture, and you may need access to a Windows Insider Program build. To determine your configuration needs and satisfy these prerequisites, see [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

Make sure you take the time to satisfy the prerequisites for GPU acceleration now. You will need to restart the installation process if you decide you want GPU acceleration during installation.

Developer tools

Prepare your target device for the installation of Azure IoT Edge for Linux on Windows and the deployment of the Linux virtual machine:

1. Set the execution policy on the target device to `AllSigned`. You can check the current execution policy in an elevated PowerShell prompt using the following command:

```
PowerShell
```

```
Get-ExecutionPolicy -List
```

If the execution policy of `local machine` is not `AllSigned`, you can set the execution policy using:

PowerShell

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Force
```

For more information on the Azure IoT Edge for Linux on Windows PowerShell module, see the [PowerShell functions reference](#).

Register your device

You can use the [Azure portal](#), [Visual Studio Code](#), or [Azure CLI](#) to register your device, depending on your preference.

Portal

In your IoT hub in the Azure portal, IoT Edge devices are created and managed separately from IoT devices that are not edge enabled.

1. Sign in to the [Azure portal](#) and navigate to your IoT hub.
2. In the left pane, select **Devices** from the menu, then select **Add Device**.
3. On the **Create a device** page, provide the following information:
 - Create a descriptive Device ID, for example `my-edge-device-1` (all lowercase). Copy this Device ID, as you'll use it later.
 - Check the **IoT Edge Device** checkbox.
 - Select **Symmetric key** as the authentication type.
 - Use the default settings to auto-generate authentication keys, which connect the new device to your hub.
4. Select **Save**.

You should see your new device listed in your IoT hub.

Now that you have a device registered in IoT Hub, you can retrieve provisioning information used to complete the installation and provisioning of the [IoT Edge runtime](#) in the next step.

View registered devices and retrieve provisioning information

Devices that use symmetric key authentication need their connection strings to complete installation and provisioning of the IoT Edge runtime. The connection string gets generated for your IoT Edge device when you create the device. For Visual Studio Code and Azure CLI, the connection string is in the JSON output. If you use the Azure portal to create your device, you can find the connection string from the device itself. When you select your device in your IoT hub, it's listed as **Primary connection string** on the device page.

Portal

The edge-enabled devices that connect to your IoT hub are listed on the **Devices** page of your IoT hub. If you have multiple devices, you can filter the list by selecting the type **IoT Edge Devices**, then select **Apply**.

When you're ready to set up your device, you need the connection string that links your physical device with its identity in the IoT hub. Devices that authenticate with symmetric keys have their connection strings available to copy in the portal. To find your connection string in the portal:

1. From the **Devices** page, select the IoT Edge device ID from the list.
2. Copy the value of either **Primary Connection String** or **Secondary Connection String**. Either key will work.

Install IoT Edge

Deploy Azure IoT Edge for Linux on Windows on your target device.

ⓘ Note

The following PowerShell process outlines how to deploy IoT Edge for Linux on Windows onto the local device. To deploy to a remote target device using PowerShell, you can use [Remote PowerShell](#) to establish a connection to a remote device and run these commands remotely on that device.

1. In an elevated PowerShell session, run either of the following commands depending on your target device architecture to download IoT Edge for Linux on

Windows.

- X64/AMD64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_X64" -OutFile
$msiPath
```

- ARM64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_ARM64" -
OutFile $msiPath
```

2. Install IoT Edge for Linux on Windows on your device.

```
PowerShell

Start-Process -Wait msieexec -ArgumentList
"/i","$([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))","/qn"
```

You can specify custom IoT Edge for Linux on Windows installation and VHDX directories by adding `INSTALLDIR=<FULLY_QUALIFIED_PATH>` and `VHDXDIR=<FULLY_QUALIFIED_PATH>` parameters to the install command. For example, if you want to use the *D:\EFLOW* folder for installation and the *D:\EFLOW-VHDX* for the VHDX, you can use the following PowerShell cmdlet.

```
PowerShell

Start-Process -Wait msieexec -ArgumentList
"/i","$([io.Path]::Combine($env:TEMP,
'AzureIoTEdge.msi'))","/qn","INSTALLDIR=D:\EFLOW", "VHDXDIR=D:\EFLOW-
VHDX"
```

3. Set the execution policy on the target device to `AllSigned` if it is not already. See the PowerShell prerequisites for commands to check the current execution policy and set the execution policy to `AllSigned`.

4. Create the IoT Edge for Linux on Windows deployment. The deployment creates your Linux virtual machine and installs the IoT Edge runtime for you.

PowerShell

`Deploy-Eflow`

💡 Tip

By default, the `Deploy-Eflow` command creates your Linux virtual machine with 1 GB of RAM, 1 vCPU core, and 16 GB of disk space. However, the resources your VM needs are highly dependent on the workloads you deploy. If your VM does not have sufficient memory to support your workloads, it will fail to start.

You can customize the virtual machine's available resources using the `Deploy-Eflow` command's optional parameters. This is required to deploy EFLOW on a device with the minimum hardware requirements.

For example, the command below creates a virtual machine with 1 vCPU core, 1 GB of RAM (represented in MB), and 2 GB of disk space:

PowerShell

```
Deploy-Eflow -cpuCount 1 -memoryInMB 1024 -vmDataSize 2
```

For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

By default, the EFLOW Linux virtual machine has no DNS configuration. Deployments using DHCP will try to obtain the DNS configuration propagated by the DHCP server. Please check your DNS configuration to ensure internet connectivity. For more information, see [AzFLOW-DNS](#).

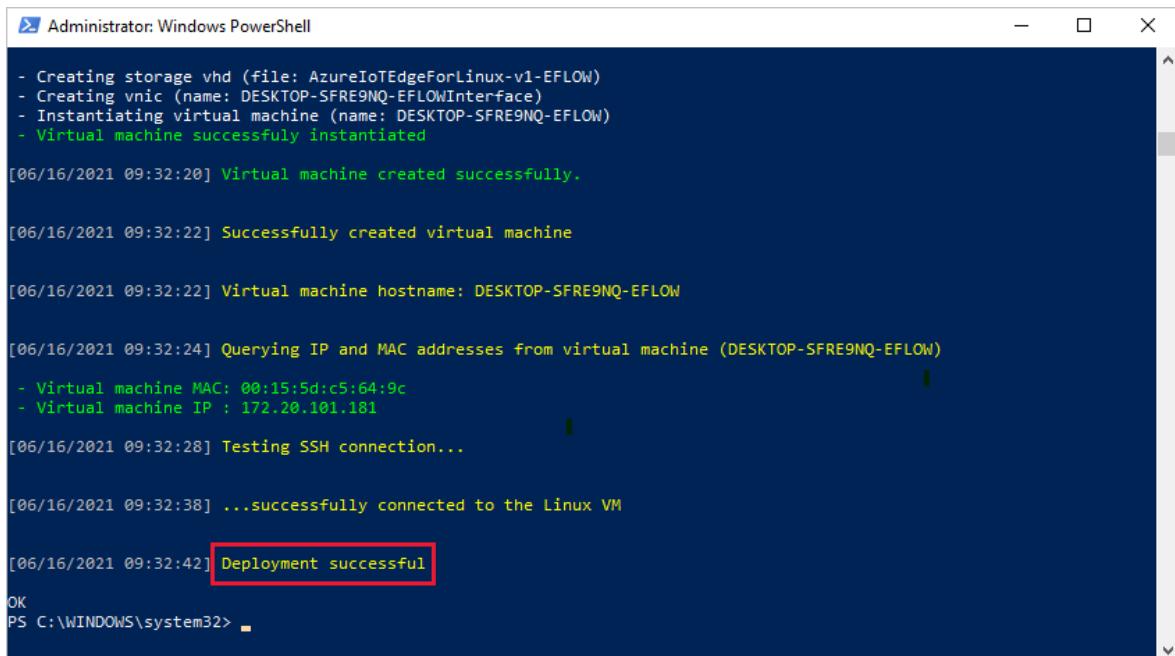
You can assign a GPU to your deployment to enable GPU-accelerated Linux modules. To gain access to these features, you will need to install the prerequisites detailed in [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

To use a GPU passthrough, add the `gpuName`, `gpuPassthroughType`, and `gpuCount` parameters to your `Deploy-Eflow` command. For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

Enabling hardware device passthrough may increase security risks. Microsoft recommends a device mitigation driver from your GPU's vendor, when applicable. For more information, see [Deploy graphics devices using discrete device assignment](#).

5. Enter 'Y' to accept the license terms.
6. Enter 'O' or 'R' to toggle **Optional diagnostic data** on or off, depending on your preference.
7. Once the deployment is complete, the PowerShell window reports **Deployment successful**.



```
- Creating storage vhd (file: AzureIoTEdgeForLinux-v1-EFLOW)
- Creating vnic (name: DESKTOP-SFRE9NQ-EFLWInterface)
- Instantiating virtual machine (name: DESKTOP-SFRE9NQ-EFLOW)
- Virtual machine successfully instantiated

[06/16/2021 09:32:20] Virtual machine created successfully.

[06/16/2021 09:32:22] Successfully created virtual machine

[06/16/2021 09:32:22] Virtual machine hostname: DESKTOP-SFRE9NQ-EFLOW

[06/16/2021 09:32:24] Querying IP and MAC addresses from virtual machine (DESKTOP-SFRE9NQ-EFLOW)
- Virtual machine MAC: 00:15:5d:c5:64:9c
- Virtual machine IP : 172.20.101.181

[06/16/2021 09:32:28] Testing SSH connection...

[06/16/2021 09:32:38] ...successfully connected to the Linux VM

[06/16/2021 09:32:42] Deployment successful
OK
PS C:\WINDOWS\system32>
```

After a successful deployment, you are ready to provision your device.

Provision the device with its cloud identity

You're ready to set up your device with its cloud identity and authentication information.

To provision your device using symmetric keys, you need your device's **connection string**.

Run the following command in an elevated PowerShell session on your target device. Replace the placeholder text with your own values.

PowerShell

```
Provision-EflowVm -provisioningType ManualConnectionString -devConnString  
"PASTE_DEVICE_CONNECTION_STRING_HERE"
```

For more information about the `Provision-EflowVM` command, see [PowerShell functions for IoT Edge for Linux on Windows](#).

Verify successful configuration

Verify that IoT Edge for Linux on Windows was successfully installed and configured on your IoT Edge device.

1. Sign in to your IoT Edge for Linux on Windows virtual machine using the following command in your PowerShell session:

PowerShell

```
Connect-EflowVm
```

 **Note**

The only account allowed to SSH to the virtual machine is the user that created it.

2. Once you are logged in, you can check the list of running IoT Edge modules using the following Linux command:

Bash

```
sudo iotedge list
```

3. If you need to troubleshoot the IoT Edge service, use the following Linux commands.

- a. Retrieve the service logs.

Bash

```
sudo iotedge system logs
```

- b. Use the `check` tool to verify configuration and connection status of the device.

Bash

```
sudo iotedge check
```

ⓘ Note

On a newly provisioned device, you may see an error related to IoT Edge Hub:

✗ production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error

Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module isn't running. To resolve the error, in IoT Hub, set the modules for the device and create a deployment. Creating a deployment for the device starts the modules on the device including the IoT Edge Hub module.

When you create a new IoT Edge device, it displays the status code `417 -- The device's deployment configuration is not set` in the Azure portal. This status is normal, and means that the device is ready to receive a module deployment.

Uninstall IoT Edge for Linux on Windows

If you want to remove the Azure IoT Edge for Linux on Windows installation from your device, use the following commands.

1. Open Settings on Windows
2. Select Add or Remove Programs
3. Select *Azure IoT Edge* app
4. Select Uninstall

Next steps

- Continue to [deploy IoT Edge modules](#) to learn how to deploy modules onto your device.

- Learn how to [manage certificates on your IoT Edge for Linux on Windows virtual machine](#) and transfer files from the host OS to your Linux virtual machine.
- Learn how to [configure your IoT Edge devices to communicate through a proxy server](#).

Create and provision IoT Edge for Linux on Windows devices at scale using X.509 certificates

Article • 06/03/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides end-to-end instructions for autoprovisioning one or more [IoT Edge for Linux on Windows](#) devices using X.509 certificates. You can automatically provision Azure IoT Edge devices with the [Azure IoT Hub device provisioning service \(DPS\)](#). If you're unfamiliar with the process of autoprovisioning, review the [provisioning overview](#) before continuing.

The tasks are as follows:

1. Generate certificates and keys.
2. Create either an **individual enrollment** for a single device or a **group enrollment** for a set of devices.
3. Deploy a Linux virtual machine with the IoT Edge runtime installed and connect it to the IoT Hub.

Using X.509 certificates as an attestation mechanism is an excellent way to scale production and simplify device provisioning. Typically, X.509 certificates are arranged in a certificate chain of trust. Starting with a self-signed or trusted root certificate, each certificate in the chain signs the next lower certificate. This pattern creates a delegated chain of trust from the root certificate down through each intermediate certificate to the final downstream device certificate installed on a device.

Prerequisites

Cloud resources

- An active IoT hub

- An instance of the IoT Hub device provisioning service in Azure, linked to your IoT hub
 - If you don't have a device provisioning service instance, you can follow the instructions in the [Create a new IoT Hub device provisioning service](#) and [Link the IoT hub and your device provisioning service](#) sections of the IoT Hub device provisioning service quickstart.
 - After you have the device provisioning service running, copy the value of **ID Scope** from the overview page. You use this value when you configure the IoT Edge runtime.

Device requirements

A Windows device with the following minimum requirements:

- System Requirements
 - Windows 10¹/11 (Pro, Enterprise, IoT Enterprise)
 - Windows Server 2019¹/2022
- Hardware requirements
 - Minimum Free Memory: 1 GB
 - Minimum Free Disk Space: 10 GB
- Virtualization support
 - On Windows 10, enable Hyper-V. For more information, see [Install Hyper-V on Windows 10](#).
 - On Windows Server, install the Hyper-V role and create a default network switch. For more information, see [Nested virtualization for Azure IoT Edge for Linux on Windows](#).
 - On a virtual machine, configure nested virtualization. For more information, see [nested virtualization](#).
- Networking support
 - Windows Server does not come with a default switch. Before you can deploy EFLOW to a Windows Server device, you need to create a virtual switch. For more information, see [Create virtual switch for Linux on Windows](#).
 - Windows Desktop versions come with a default switch that can be used for EFLOW installation. If needed, you can create your own custom virtual switch.

 Tip

If you want to use **GPU-accelerated Linux modules** in your Azure IoT Edge for Linux on Windows deployment, there are several configuration options to consider.

You will need to install the correct drivers depending on your GPU architecture, and you may need access to a Windows Insider Program build. To determine your configuration needs and satisfy these prerequisites, see [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

Make sure you take the time to satisfy the prerequisites for GPU acceleration now. You will need to restart the installation process if you decide you want GPU acceleration during installation.

Developer tools

Prepare your target device for the installation of Azure IoT Edge for Linux on Windows and the deployment of the Linux virtual machine:

1. Set the execution policy on the target device to `AllSigned`. You can check the current execution policy in an elevated PowerShell prompt using the following command:

```
PowerShell
```

```
Get-ExecutionPolicy -List
```

If the execution policy of `local machine` is not `AllSigned`, you can set the execution policy using:

```
PowerShell
```

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Force
```

For more information on the Azure IoT Edge for Linux on Windows PowerShell module, see the [PowerShell functions reference](#).

Generate device identity certificates

The device identity certificate is a downstream certificate that connects through a certificate chain of trust to the top X.509 certificate authority (CA) certificate. The device identity certificate must have its common name (CN) set to the device ID that you want the device to have in your IoT hub.

Device identity certificates are only used for provisioning the IoT Edge device and authenticating the device with Azure IoT Hub. They aren't signing certificates, unlike the CA certificates that the IoT Edge device presents to modules or downstream devices for verification. For more information, see [Azure IoT Edge certificate usage detail](#).

After you create the device identity certificate, you should have two files: a .cer or .pem file that contains the public portion of the certificate, and a .cer or .pem file with the private key of the certificate. If you plan to use group enrollment in DPS, you also need the public portion of an intermediate or root CA certificate in the same certificate chain of trust.

You need the following files to set up automatic provisioning with X.509:

- The device identity certificate and its private key certificate. The device identity certificate is uploaded to DPS if you create an individual enrollment. The private key is passed to the IoT Edge runtime.
- A full chain certificate, which should have at least the device identity and the intermediate certificates in it. The full chain certificate is passed to the IoT Edge runtime.
- An intermediate or root CA certificate from the certificate chain of trust. This certificate is uploaded to DPS if you create a group enrollment.

 **Note**

Currently, a limitation in libiothsm prevents the use of certificates that expire on or after January 1, 2038.

Use test certificates (optional)

If you don't have a certificate authority available to create new identity certs and want to try out this scenario, the Azure IoT Edge git repository contains scripts that you can use to generate test certificates. These certificates are designed for development testing only, and must not be used in production.

To create test certificates, follow the steps in [Create demo certificates to test IoT Edge device features](#). Complete the two required sections to set up the certificate generation scripts and to create a root CA certificate. Then, follow the steps to create a device identity certificate. When you're finished, you should have the following certificate chain and key pair:

- <WRKDIR>\certs\iot-edge-device-identity-<name>-full-chain.cert.pem
- <WRKDIR>\private\iot-edge-device-identity-<name>.key.pem

You need both these certificates on the IoT Edge device. If you're going to use individual enrollment in DPS, then you upload the .cert.pem file. If you're going to use group enrollment in DPS, then you also need an intermediate or root CA certificate in the same certificate chain of trust to upload. If you're using demo certs, use the `<WRKDIR>\certs\azure-iot-test-only.root.ca.cert.pem` certificate for group enrollment.

Create a DPS enrollment

Use your generated certificates and keys to create an enrollment in DPS for one or more IoT Edge devices.

If you are looking to provision a single IoT Edge device, create an **individual enrollment**. If you need multiple devices provisioned, follow the steps for creating a **DPS group enrollment**.

When you create an enrollment in DPS, you have the opportunity to declare an **Initial Device Twin State**. In the device twin, you can set tags to group devices by any metric you need in your solution, like region, environment, location, or device type. These tags are used to create [automatic deployments](#).

For more information about enrollments in the device provisioning service, see [How to manage device enrollments](#).

Individual enrollment

Create a DPS individual enrollment

Individual enrollments take the public portion of a device's identity certificate and match that to the certificate on the device.

Tip

The steps in this article are for the Azure portal, but you can also create individual enrollments using the Azure CLI. For more information, see [az iot dps enrollment](#). As part of the CLI command, use the `edge-enabled` flag to specify that the enrollment is for an IoT Edge device.

1. In the [Azure portal](#), navigate to your instance of IoT Hub device provisioning service.

2. Under **Settings**, select **Manage enrollments**.
3. Select **Add individual enrollment** then complete the following steps to configure the enrollment:
 - **Mechanism:** Select **X.509**.
 - **Primary Certificate .pem or .cer file:** Upload the public file from the device identity certificate. If you used the scripts to generate a test certificate, choose the following file:
`<WRKDIR>\certs\iot-edge-device-identity-<name>.cert.pem`
 - **IoT Hub Device ID:** Provide an ID for your device if you'd like. You can use device IDs to target an individual device for module deployment. If you don't provide a device ID, the common name (CN) in the X.509 certificate is used.
 - **IoT Edge device:** Select **True** to declare that the enrollment is for an IoT Edge device.
 - **Select the IoT hubs this device can be assigned to:** Choose the linked IoT hub that you want to connect your device to. You can choose multiple hubs, and the device will be assigned to one of them according to the selected allocation policy.
 - **Initial Device Twin State:** Add a tag value to be added to the device twin if you'd like. You can use tags to target groups of devices for automatic deployment. For example:

```
JSON

{
  "tags": {
    "environment": "test"
  },
  "properties": {
    "desired": {}
  }
}
```

4. Select **Save**.

Under **Manage Enrollments**, you can see the **Registration ID** for the enrollment you just created. Make note of it, as it can be used when you provision your device.

Now that an enrollment exists for this device, the IoT Edge runtime can automatically provision the device during installation.

Install IoT Edge

Deploy Azure IoT Edge for Linux on Windows on your target device.

ⓘ Note

The following PowerShell process outlines how to deploy IoT Edge for Linux on Windows onto the local device. To deploy to a remote target device using PowerShell, you can use [Remote PowerShell](#) to establish a connection to a remote device and run these commands remotely on that device.

1. In an elevated PowerShell session, run either of the following commands depending on your target device architecture to download IoT Edge for Linux on Windows.

- X64/AMD64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_X64" -OutFile
$msiPath
```

- ARM64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_ARM64" -
OutFile $msiPath
```

2. Install IoT Edge for Linux on Windows on your device.

```
PowerShell

Start-Process -Wait msieexec -ArgumentList
"/i","$([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))","/qn"
```

You can specify custom IoT Edge for Linux on Windows installation and VHDX directories by adding `INSTALLDIR=<FULLY_QUALIFIED_PATH>` and `VHDXDIR=<FULLY_QUALIFIED_PATH>` parameters to the install command. For example, if you want to use the *D:\EFLOW* folder for installation and the *D:\EFLOW-VHDX* for the VHDX, you can use the following PowerShell cmdlet.

PowerShell

```
Start-Process -Wait msixexec -ArgumentList  
"/i",$([io.Path]::Combine($env:TEMP,  
'AzureIoTEdge.msi'))","/qn","INSTALLDIR=D:\EFLOW", "VHDXDIR=D:\EFLOW-  
VHDX"
```

3. Set the execution policy on the target device to `AllSigned` if it is not already. See the PowerShell prerequisites for commands to check the current execution policy and set the execution policy to `AllSigned`.
4. Create the IoT Edge for Linux on Windows deployment. The deployment creates your Linux virtual machine and installs the IoT Edge runtime for you.

PowerShell

[Deploy-Eflow](#)

💡 Tip

By default, the `Deploy-Eflow` command creates your Linux virtual machine with 1 GB of RAM, 1 vCPU core, and 16 GB of disk space. However, the resources your VM needs are highly dependent on the workloads you deploy. If your VM does not have sufficient memory to support your workloads, it will fail to start.

You can customize the virtual machine's available resources using the `Deploy-Eflow` command's optional parameters. This is required to deploy EFLOW on a device with the minimum hardware requirements.

For example, the command below creates a virtual machine with 1 vCPU core, 1 GB of RAM (represented in MB), and 2 GB of disk space:

PowerShell

```
Deploy-Eflow -cpuCount 1 -memoryInMB 1024 -vmDataSize 2
```

For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

By default, the EFLOW Linux virtual machine has no DNS configuration. Deployments using DHCP will try to obtain the DNS configuration propagated by the DHCP server. Please check your DNS configuration to ensure internet connectivity. For more information, see [AzEFLOW-DNS](#).

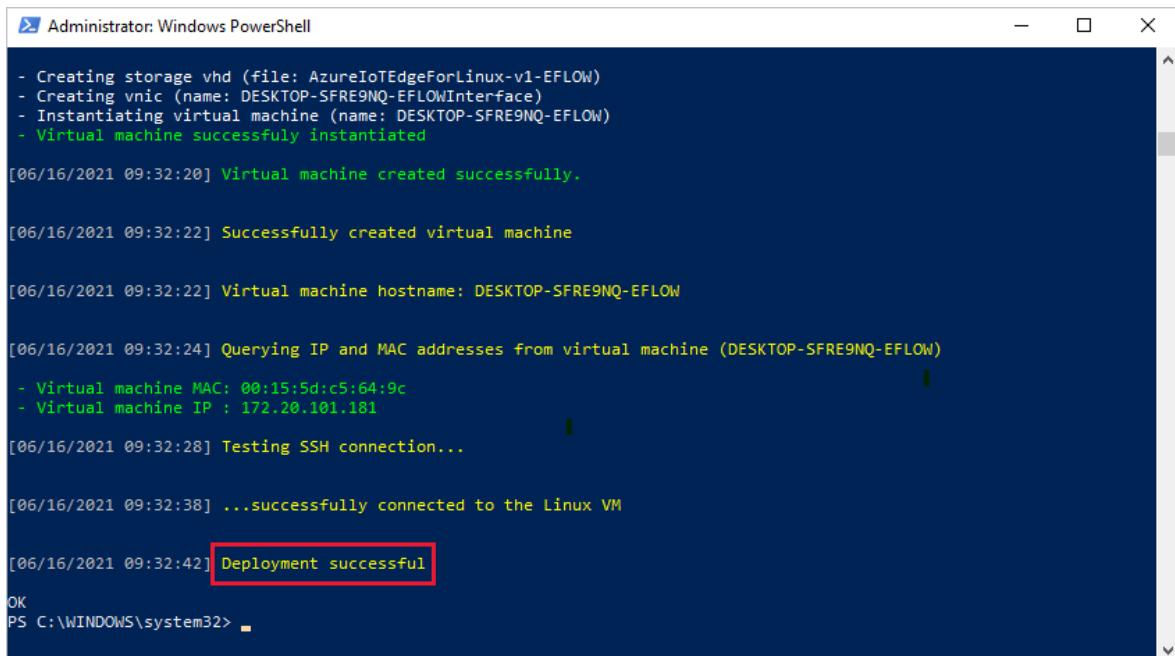
You can assign a GPU to your deployment to enable GPU-accelerated Linux modules. To gain access to these features, you will need to install the prerequisites detailed in [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

To use a GPU passthrough, add the `gpuName`, `gpuPassthroughType`, and `gpuCount` parameters to your `Deploy-Eflow` command. For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

Enabling hardware device passthrough may increase security risks. Microsoft recommends a device mitigation driver from your GPU's vendor, when applicable. For more information, see [Deploy graphics devices using discrete device assignment](#).

5. Enter 'Y' to accept the license terms.
6. Enter 'O' or 'R' to toggle **Optional diagnostic data** on or off, depending on your preference.
7. Once the deployment is complete, the PowerShell window reports **Deployment successful**.



The screenshot shows a Windows PowerShell window titled "Administrator: Windows PowerShell". The console output details the deployment process:

- Creating storage vhd (file: AzureIoTEdgeForLinux-v1-EFLOW)
- Creating vnic (name: DESKTOP-SFRE9NQ-EFLOWInterface)
- Instantiating virtual machine (name: DESKTOP-SFRE9NQ-EFLOW)
- Virtual machine successfully instantiated

[06/16/2021 09:32:20] Virtual machine created successfully.

[06/16/2021 09:32:22] Successfully created virtual machine

[06/16/2021 09:32:22] Virtual machine hostname: DESKTOP-SFRE9NQ-EFLOW

[06/16/2021 09:32:24] Querying IP and MAC addresses from virtual machine (DESKTOP-SFRE9NQ-EFLOW)

- Virtual machine MAC: 00:15:5d:c5:64:9c
- Virtual machine IP : 172.20.101.181

[06/16/2021 09:32:28] Testing SSH connection...

[06/16/2021 09:32:38] ...successfully connected to the Linux VM

[06/16/2021 09:32:42] Deployment successful

OK
PS C:\WINDOWS\system32> ■

After a successful deployment, you are ready to provision your device.

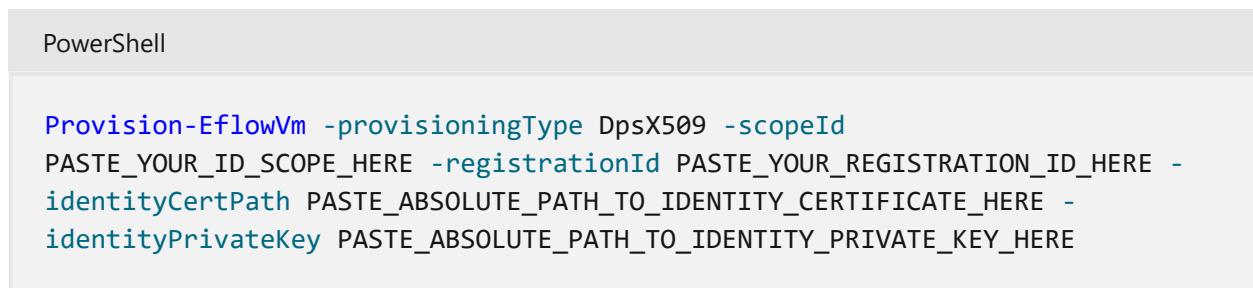
Provision the device with its cloud identity

Once the runtime is installed on your device, configure the device with the information it uses to connect to the device provisioning service and IoT Hub.

Have the following information ready:

- The DPS ID Scope value. You can retrieve this value from the overview page of your DPS instance in the Azure portal.
- The device identity certificate chain file on the device.
- The device identity key file on the device.

Run the following command in an elevated PowerShell session with the placeholder values updated with your own values:



PowerShell

```
Provision-EflowVm -provisioningType DpsX509 -scopeId
PASTE_YOUR_ID_SCOPE_HERE -registrationId PASTE_YOUR_REGISTRATION_ID_HERE -
identityCertPath PASTE_ABSOLUTE_PATH_TO_IDENTITY_CERTIFICATE_HERE -
identityPrivateKey PASTE_ABSOLUTE_PATH_TO_IDENTITY_PRIVATE_KEY_HERE
```

Verify successful installation

Verify that IoT Edge for Linux on Windows was successfully installed and configured on your IoT Edge device.

Individual enrollment

You can verify that the individual enrollment that you created in device provisioning service was used. Navigate to your device provisioning service instance in the Azure portal. Open the enrollment details for the individual enrollment that you created. Notice that the status of the enrollment is **assigned** and the device ID is listed.

1. Sign in to your IoT Edge for Linux on Windows virtual machine using the following command in your PowerShell session:

```
PowerShell
```

```
Connect-EflowVm
```

ⓘ Note

The only account allowed to SSH to the virtual machine is the user that created it.

2. Once you are logged in, you can check the list of running IoT Edge modules using the following Linux command:

```
Bash
```

```
sudo iotedge list
```

3. If you need to troubleshoot the IoT Edge service, use the following Linux commands.

- a. If you need to troubleshoot the service, retrieve the service logs.

```
Bash
```

```
sudo iotedge system logs
```

- b. Use the `check` tool to verify configuration and connection status of the device.

```
Bash
```

```
sudo iotedge check
```

Note

On a newly provisioned device, you may see an error related to IoT Edge Hub:

- ✗ **production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error**

Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module isn't running. To resolve the error, in IoT Hub, set the modules for the device and create a deployment. Creating a deployment for the device starts the modules on the device including the IoT Edge Hub module.

When you create a new IoT Edge device, it displays the status code `417 -- The device's deployment configuration is not set` in the Azure portal. This status is normal, and means that the device is ready to receive a module deployment.

Uninstall IoT Edge for Linux on Windows

If you want to remove the Azure IoT Edge for Linux on Windows installation from your device, use the following commands.

1. Open Settings on Windows
2. Select Add or Remove Programs
3. Select *Azure IoT Edge* app
4. Select Uninstall

Next steps

The device provisioning service enrollment process lets you set the device ID and device twin tags at the same time as you provision the new device. You can use those values to target individual devices or groups of devices using automatic device management.

Learn how to [Deploy and monitor IoT Edge modules at scale using the Azure portal](#) or [using Azure CLI](#).

You can also:

- Continue to [deploy IoT Edge modules](#) to learn how to deploy modules onto your device.

- Learn how to [manage certificates on your IoT Edge for Linux on Windows virtual machine](#) and transfer files from the host OS to your Linux virtual machine.
- Learn how to [configure your IoT Edge devices to communicate through a proxy server](#).

Create and provision an IoT Edge for Linux on Windows device at scale by using a TPM

Article • 06/03/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article provides instructions for autoprovisioning an Azure IoT Edge for Linux on Windows device by using a Trusted Platform Module (TPM). You can automatically provision Azure IoT Edge devices with the [Azure IoT Hub device provisioning service](#). If you're unfamiliar with the process of autoprovisioning, review the [provisioning overview](#) before you continue.

This article outlines two methodologies. Select your preference based on the architecture of your solution:

- Autoprovision a Linux on Windows device with physical TPM hardware.
- Autoprovision a Linux on Windows device by using a simulated TPM. We recommend this methodology only as a testing scenario. A simulated TPM doesn't offer the same security as a physical TPM.

The tasks are as follows:

Physical TPM

- Install IoT Edge for Linux on Windows.
- Retrieve the TPM information from your device.
- Create an individual enrollment for the device.
- Provision your device with its TPM information.

Prerequisites

Cloud resources

- An active IoT hub
- An instance of the IoT Hub device provisioning service in Azure, linked to your IoT hub
 - If you don't have a device provisioning service instance, you can follow the instructions in the [Create a new IoT Hub device provisioning service](#) and [Link the IoT hub and your device provisioning service](#) sections of the IoT Hub device provisioning service quickstart.
 - After you have the device provisioning service running, copy the value of **ID Scope** from the overview page. You use this value when you configure the IoT Edge runtime.

Device requirements

A Windows device with the following minimum requirements:

- System Requirements
 - Windows 10¹/11 (Pro, Enterprise, IoT Enterprise)
 - Windows Server 2019¹/2022
- Hardware requirements
 - Minimum Free Memory: 1 GB
 - Minimum Free Disk Space: 10 GB
- Virtualization support
 - On Windows 10, enable Hyper-V. For more information, see [Install Hyper-V on Windows 10](#).
 - On Windows Server, install the Hyper-V role and create a default network switch. For more information, see [Nested virtualization for Azure IoT Edge for Linux on Windows](#).
 - On a virtual machine, configure nested virtualization. For more information, see [nested virtualization](#).
- Networking support
 - Windows Server does not come with a default switch. Before you can deploy EFLOW to a Windows Server device, you need to create a virtual switch. For more information, see [Create virtual switch for Linux on Windows](#).
 - Windows Desktop versions come with a default switch that can be used for EFLOW installation. If needed, you can create your own custom virtual switch.

¹ Windows 10 and Windows Server 2019 minimum build 17763 with all current cumulative updates installed.

💡 Tip

If you want to use **GPU-accelerated Linux modules** in your Azure IoT Edge for Linux on Windows deployment, there are several configuration options to consider.

You will need to install the correct drivers depending on your GPU architecture, and you may need access to a Windows Insider Program build. To determine your configuration needs and satisfy these prerequisites, see [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

Make sure you take the time to satisfy the prerequisites for GPU acceleration now. You will need to restart the installation process if you decide you want GPU acceleration during installation.

Developer tools

Prepare your target device for the installation of Azure IoT Edge for Linux on Windows and the deployment of the Linux virtual machine:

1. Set the execution policy on the target device to `AllSigned`. You can check the current execution policy in an elevated PowerShell prompt using the following command:

```
PowerShell
```

```
Get-ExecutionPolicy -List
```

If the execution policy of `local machine` is not `AllSigned`, you can set the execution policy using:

```
PowerShell
```

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Force
```

For more information on the Azure IoT Edge for Linux on Windows PowerShell module, see the [PowerShell functions reference](#).

ⓘ Note

TPM 2.0 is required when you use TPM attestation with the device provisioning service.

You can only create individual, not group, device provisioning service enrollments when you use a TPM.

Install IoT Edge

Deploy Azure IoT Edge for Linux on Windows on your target device.

ⓘ Note

The following PowerShell process outlines how to deploy IoT Edge for Linux on Windows onto the local device. To deploy to a remote target device using PowerShell, you can use [Remote PowerShell](#) to establish a connection to a remote device and run these commands remotely on that device.

1. In an elevated PowerShell session, run either of the following commands depending on your target device architecture to download IoT Edge for Linux on Windows.

- X64/AMD64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_X64" -OutFile
$msiPath
```

- ARM64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_ARM64" -
OutFile $msiPath
```

2. Install IoT Edge for Linux on Windows on your device.

```
PowerShell

Start-Process -Wait msieexec -ArgumentList
"/i","$([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))","/qn"
```

You can specify custom IoT Edge for Linux on Windows installation and VHDX directories by adding `INSTALLDIR=<FULLY_QUALIFIED_PATH>` and `VHDXDIR=<FULLY_QUALIFIED_PATH>` parameters to the install command. For example, if you want to use the *D:\EFLOW* folder for installation and the *D:\EFLOW-VHDX* for the VHDX, you can use the following PowerShell cmdlet.

PowerShell

```
Start-Process -Wait msieexec -ArgumentList  
"/i",$([io.Path]::Combine($env:TEMP,  
'AzureIoTEdge.msi'))","/qn","INSTALLDIR=D:\EFLOW", "VHDXDIR=D:\EFLOW-  
VHDX"
```

3. Set the execution policy on the target device to `AllSigned` if it is not already. See the PowerShell prerequisites for commands to check the current execution policy and set the execution policy to `AllSigned`.
4. Create the IoT Edge for Linux on Windows deployment. The deployment creates your Linux virtual machine and installs the IoT Edge runtime for you.

PowerShell

[Deploy-Eflow](#)

💡 Tip

By default, the `Deploy-Eflow` command creates your Linux virtual machine with 1 GB of RAM, 1 vCPU core, and 16 GB of disk space. However, the resources your VM needs are highly dependent on the workloads you deploy. If your VM does not have sufficient memory to support your workloads, it will fail to start.

You can customize the virtual machine's available resources using the `Deploy-Eflow` command's optional parameters. This is required to deploy EFLOW on a device with the minimum hardware requirements.

For example, the command below creates a virtual machine with 1 vCPU core, 1 GB of RAM (represented in MB), and 2 GB of disk space:

PowerShell

```
Deploy-Eflow -cpuCount 1 -memoryInMB 1024 -vmDataSize 2
```

For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

By default, the EFLOW Linux virtual machine has no DNS configuration. Deployments using DHCP will try to obtain the DNS configuration propagated by the DHCP server. Please check your DNS configuration to ensure internet connectivity. For more information, see [AzEFLow-DNS](#).

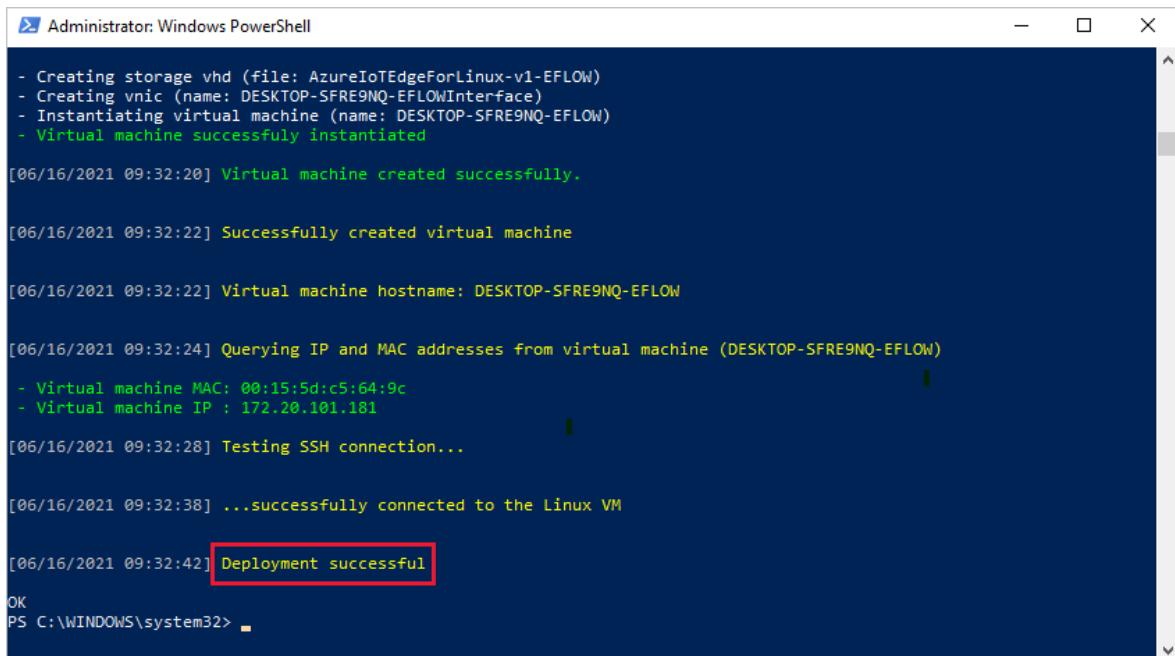
You can assign a GPU to your deployment to enable GPU-accelerated Linux modules. To gain access to these features, you will need to install the prerequisites detailed in [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

To use a GPU passthrough, add the `gpuName`, `gpuPassthroughType`, and `gpuCount` parameters to your `Deploy-Eflow` command. For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

Enabling hardware device passthrough may increase security risks. Microsoft recommends a device mitigation driver from your GPU's vendor, when applicable. For more information, see [Deploy graphics devices using discrete device assignment](#).

5. Enter 'Y' to accept the license terms.
6. Enter 'O' or 'R' to toggle **Optional diagnostic data** on or off, depending on your preference.
7. Once the deployment is complete, the PowerShell window reports **Deployment successful**.



The screenshot shows an Administrator: Windows PowerShell window. The logs indicate the creation of storage vhd, instantiation of the virtual machine, successful creation of the virtual machine, querying of IP and MAC addresses, testing of SSH connection, and finally a message stating "Deployment successful". The "Deployment successful" message is highlighted with a red rectangle.

```
- Creating storage vhd (file: AzureIoTEdgeForLinux-v1-EFLOW)
- Creating vnic (name: DESKTOP-SFRE9NQ-EFLOWInterface)
- Instantiating virtual machine (name: DESKTOP-SFRE9NQ-EFLOW)
- Virtual machine successfully instantiated

[06/16/2021 09:32:20] Virtual machine created successfully.

[06/16/2021 09:32:22] Successfully created virtual machine

[06/16/2021 09:32:22] Virtual machine hostname: DESKTOP-SFRE9NQ-EFLOW

[06/16/2021 09:32:24] Querying IP and MAC addresses from virtual machine (DESKTOP-SFRE9NQ-EFLOW)
- Virtual machine MAC: 00:15:5d:c5:64:9c
- Virtual machine IP : 172.20.101.181

[06/16/2021 09:32:28] Testing SSH connection...

[06/16/2021 09:32:38] ...successfully connected to the Linux VM

[06/16/2021 09:32:42] Deployment successful
OK
PS C:\WINDOWS\system32> ■
```

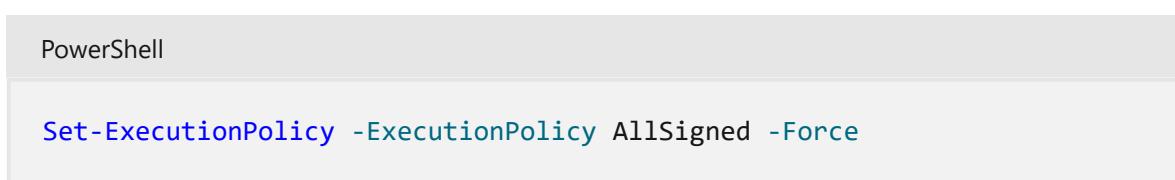
After a successful deployment, you are ready to provision your device.

There are some steps to prepare your device for provisioning with TPM. Leave your deployment open while you prepare your device. You'll return to your deployment later in the article.

Enable TPM passthrough

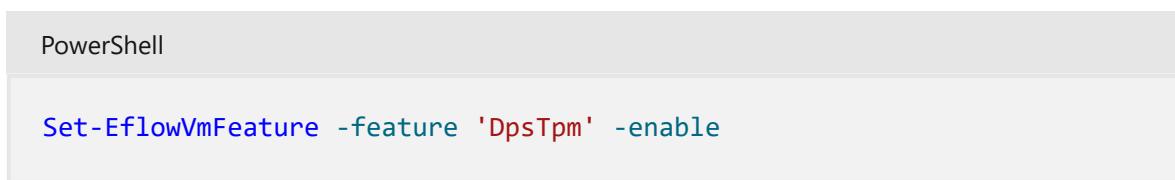
The IoT Edge for Linux on Windows VM has a TPM feature that can be enabled or disabled. By default, it's disabled. When this feature is enabled, the VM can access the host machine's TPM.

1. Open PowerShell in an elevated session.
2. If you haven't already, set the execution policy on your device to `AllSigned` so that you can run the IoT Edge for Linux on Windows PowerShell functions.



```
PowerShell
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Force
```

3. Turn on the TPM feature.



```
PowerShell
Set-EflowVmFeature -feature 'DpsTpm' -enable
```

Retrieve the TPM information from your device

Physical TPM

To provision your device, you need an **Endorsement key** for your TPM chip and **Registration ID** for your device. You provide this information to your instance of the device provisioning service so that the service can recognize your device when it tries to connect.

The endorsement key is unique to each TPM chip. It is obtained from the TPM chip manufacturer associated with it. You can derive a unique registration ID for your TPM device by, for example, creating an SHA-256 hash of the endorsement key.

IoT Edge for Linux on Windows provides a PowerShell script to help retrieve this information from your TPM. To use the script, follow these steps on your device:

1. Open PowerShell in an elevated session.
2. Run the command.

PowerShell

```
Get-EflowVmTpmProvisioningInfo | Format-List
```

Create a device provisioning service enrollment

Use your TPM's provisioning information to create an individual enrollment in the device provisioning service.

When you create an enrollment in the device provisioning service, you have the opportunity to declare an **Initial Device Twin State**. In the device twin, you can set tags to group devices by any metric used in your solution, like region, environment, location, or device type. These tags are used to create [automatic deployments](#).

Tip

The steps in this article are for the Azure portal, but you can also create individual enrollments by using the Azure CLI. For more information, see [az iot dps enrollment](#). As part of the CLI command, use the `edge-enabled` flag to specify that the enrollment is for an IoT Edge device.

1. In the [Azure portal](#), go to your instance of the IoT Hub device provisioning service.
2. Under **Settings**, select **Manage enrollments**.
3. Select **Add individual enrollment**, and then complete the following steps to configure the enrollment:
 - a. For **Mechanism**, select **TPM**.
 - b. Provide the **Endorsement key** and **Registration ID** that you copied from your VM or physical device.
 - c. Provide an ID for your device if you want. If you don't provide a device ID, the registration ID is used.
 - d. Select **True** to declare that your VM or physical device is an IoT Edge device.
 - e. Choose the linked IoT hub that you want to connect your device to, or select **Link to new IoT Hub**. You can choose multiple hubs, and the device will be assigned to one of them according to the selected assignment policy.
 - f. Add a tag value to the **Initial Device Twin State** if you want. You can use tags to target groups of devices for module deployment. For more information, see [Deploy IoT Edge modules at scale](#).
 - g. Select **Save**.

Now that an enrollment exists for this device, the IoT Edge runtime can automatically provision the device during installation.

Provision the device with its cloud identity

1. Open an elevated PowerShell session on the Windows device.
2. Provision your device by using the **Scope ID** that you collected from your instance of the device provisioning service.

```
PowerShell
```

```
Provision-EflowVM -provisioningType "DpsTpm" -scopeId "SCOPE_ID_HERE"
```

If you enrolled the device using a custom **Registration Id**, you must specify that registration ID as well when provisioning:

PowerShell

```
Provision-EflowVM -provisioningType "DpsTpm" -scopeId "SCOPE_ID_HERE" -  
registrationId "REGISTRATION_ID_HERE"
```

Verify successful installation

Verify that IoT Edge for Linux on Windows was successfully installed and configured on your IoT Edge device.

If the runtime started successfully, you can go into your IoT hub and start deploying IoT Edge modules to your device.

You can verify that the individual enrollment that you created in the device provisioning service was used. Go to your device provisioning service instance in the Azure portal. Open the enrollment details for the individual enrollment that you created. Notice that the status of the enrollment is **assigned** and the device ID is listed.

Use the following commands on your device to verify that the IoT Edge installed and started successfully.

1. Connect to your IoT Edge for Linux on Windows VM by using the following command in your PowerShell session:

PowerShell

```
Connect-EflowVm
```

 **Note**

The only account allowed to SSH to the VM is the user who created it.

2. After you're signed in, you can check the list of running IoT Edge modules by using the following Linux command:

Bash

```
sudo iotedge list
```

3. If you need to troubleshoot the IoT Edge service, use the following Linux commands.

- a. If you need to troubleshoot the service, retrieve the service logs.

```
Bash
```

```
sudo iotedge system logs
```

- b. Use the `check` tool to verify configuration and connection status of the device.

```
Bash
```

```
sudo iotedge check
```

⚠ Note

On a newly provisioned device, you may see an error related to IoT Edge Hub:

✗ production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error

Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module isn't running. To resolve the error, in IoT Hub, set the modules for the device and create a deployment. Creating a deployment for the device starts the modules on the device including the IoT Edge Hub module.

Uninstall IoT Edge for Linux on Windows

If you want to remove the Azure IoT Edge for Linux on Windows installation from your device, use the following commands.

1. Open Settings on Windows
2. Select Add or Remove Programs
3. Select *Azure IoT Edge* app
4. Select Uninstall

Next steps

The device provisioning service enrollment process lets you set the device ID and device twin tags at the same time as you provision the new device. You can use those values to target individual devices or groups of devices by using automatic device management.

Learn how to [deploy](#) and monitor IoT Edge modules at scale by using the Azure portal or [the Azure CLI](#).

Create and provision IoT Edge for Linux on Windows devices at scale using symmetric keys

Article • 06/03/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article shows how to autoprovision one or more [IoT Edge for Linux on Windows](#) devices using symmetric keys. You can automatically provision Azure IoT Edge devices with the [Azure IoT Hub device provisioning service \(DPS\)](#). If you're unfamiliar with the process of autoprovisioning, review the [provisioning overview](#) before continuing.

The tasks are as follows:

1. Create either an [individual enrollment](#) for a single device or a [group enrollment](#) for a set of devices.
2. Deploy a Linux virtual machine with the IoT Edge runtime installed and connect it to the IoT Hub.

Symmetric key attestation is a simple approach to authenticating a device with a device provisioning service instance. This attestation method represents a "Hello world" experience for developers who are new to device provisioning, or do not have strict security requirements. Device attestation using a [TPM](#) or [X.509 certificates](#) is more secure, and should be used for more stringent security requirements.

Prerequisites

Cloud resources

- An active IoT hub
- An instance of the IoT Hub device provisioning service in Azure, linked to your IoT hub

- If you don't have a device provisioning service instance, you can follow the instructions in the [Create a new IoT Hub device provisioning service](#) and [Link the IoT hub and your device provisioning service](#) sections of the IoT Hub device provisioning service quickstart.
- After you have the device provisioning service running, copy the value of **ID Scope** from the overview page. You use this value when you configure the IoT Edge runtime.

Device requirements

A Windows device with the following minimum requirements:

- System Requirements
 - Windows 10¹/11 (Pro, Enterprise, IoT Enterprise)
 - Windows Server 2019¹/2022
- Hardware requirements
 - Minimum Free Memory: 1 GB
 - Minimum Free Disk Space: 10 GB
- Virtualization support
 - On Windows 10, enable Hyper-V. For more information, see [Install Hyper-V on Windows 10](#).
 - On Windows Server, install the Hyper-V role and create a default network switch. For more information, see [Nested virtualization for Azure IoT Edge for Linux on Windows](#).
 - On a virtual machine, configure nested virtualization. For more information, see [nested virtualization](#).
- Networking support
 - Windows Server does not come with a default switch. Before you can deploy EFLOW to a Windows Server device, you need to create a virtual switch. For more information, see [Create virtual switch for Linux on Windows](#).
 - Windows Desktop versions come with a default switch that can be used for EFLOW installation. If needed, you can create your own custom virtual switch.

💡 Tip

If you want to use **GPU-accelerated Linux modules** in your Azure IoT Edge for Linux on Windows deployment, there are several configuration options to consider.

You will need to install the correct drivers depending on your GPU architecture, and you may need access to a Windows Insider Program build. To determine your configuration needs and satisfy these prerequisites, see [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

Make sure you take the time to satisfy the prerequisites for GPU acceleration now. You will need to restart the installation process if you decide you want GPU acceleration during installation.

Developer tools

Prepare your target device for the installation of Azure IoT Edge for Linux on Windows and the deployment of the Linux virtual machine:

1. Set the execution policy on the target device to `AllSigned`. You can check the current execution policy in an elevated PowerShell prompt using the following command:

```
PowerShell
```

```
Get-ExecutionPolicy -List
```

If the execution policy of `local machine` is not `AllSigned`, you can set the execution policy using:

```
PowerShell
```

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Force
```

For more information on the Azure IoT Edge for Linux on Windows PowerShell module, see the [PowerShell functions reference](#).

Create a DPS enrollment

Create an enrollment to provision one or more devices through DPS.

If you are looking to provision a single IoT Edge device, create an [individual enrollment](#). If you need multiple devices provisioned, follow the steps for creating a [DPS group enrollment](#).

When you create an enrollment in DPS, you have the opportunity to declare an [initial device twin state](#). In the device twin, you can set tags to group devices by any metric

you need in your solution, like region, environment, location, or device type. These tags are used to create [automatic deployments](#).

For more information about enrollments in the device provisioning service, see [How to manage device enrollments](#).

Individual enrollment

Create a DPS individual enrollment

Tip

The steps in this article are for the Azure portal, but you can also create individual enrollments using the Azure CLI. For more information, see [az iot dps enrollment](#). As part of the CLI command, use the **edge-enabled** flag to specify that the enrollment is for an IoT Edge device.

1. In the [Azure portal](#), navigate to your instance of IoT Hub device provisioning service.
2. Under **Settings**, select **Manage enrollments**.
3. Select **Add individual enrollment** then complete the following steps to configure the enrollment:
 - a. For **Mechanism**, select **Symmetric Key**.
 - b. Provide a unique **Registration ID** for your device.
 - c. Optionally, provide an **IoT Hub Device ID** for your device. You can use device IDs to target an individual device for module deployment. If you don't provide a device ID, the registration ID is used.
 - d. Select **True** to declare that the enrollment is for an IoT Edge device.
 - e. Optionally, add a tag value to the **Initial Device Twin State**. You can use tags to target groups of devices for module deployment. For example:

JSON

```
{  
  "tags": {  
    "environment": "test"  
  }
```

```
        },
        "properties": {
            "desired": {}
        }
    }
}
```

f. Select **Save**.

4. Copy the individual enrollment's **Primary Key** value to use when installing the IoT Edge runtime.

Now that an enrollment exists for this device, the IoT Edge runtime can automatically provision the device during installation.

Install IoT Edge

Deploy Azure IoT Edge for Linux on Windows on your target device.

ⓘ Note

The following PowerShell process outlines how to deploy IoT Edge for Linux on Windows onto the local device. To deploy to a remote target device using PowerShell, you can use [Remote PowerShell](#) to establish a connection to a remote device and run these commands remotely on that device.

1. In an elevated PowerShell session, run either of the following commands depending on your target device architecture to download IoT Edge for Linux on Windows.

- X64/AMD64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_X64" -OutFile
$msiPath
```

- ARM64

```
PowerShell

$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))
$ProgressPreference = 'SilentlyContinue'
```

```
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_ARM64" -  
OutFile $msiPath
```

2. Install IoT Edge for Linux on Windows on your device.

PowerShell

```
Start-Process -Wait msiexec -ArgumentList  
"/i","$([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))","/qn"
```

You can specify custom IoT Edge for Linux on Windows installation and VHDX directories by adding `INSTALLDIR=<FULLY_QUALIFIED_PATH>` and `VHDXDIR=<FULLY_QUALIFIED_PATH>` parameters to the install command. For example, if you want to use the *D:\EFLOW* folder for installation and the *D:\EFLOW-VHDX* for the VHDX, you can use the following PowerShell cmdlet.

PowerShell

```
Start-Process -Wait msiexec -ArgumentList  
"/i","$([io.Path]::Combine($env:TEMP,  
'AzureIoTEdge.msi'))","/qn","INSTALLDIR=D:\EFLOW", "VHDXDIR=D:\EFLOW-  
VHDX"
```

3. Set the execution policy on the target device to `AllSigned` if it is not already. See the PowerShell prerequisites for commands to check the current execution policy and set the execution policy to `AllSigned`.
4. Create the IoT Edge for Linux on Windows deployment. The deployment creates your Linux virtual machine and installs the IoT Edge runtime for you.

PowerShell

[Deploy-Eflow](#)

Tip

By default, the `Deploy-Eflow` command creates your Linux virtual machine with 1 GB of RAM, 1 vCPU core, and 16 GB of disk space. However, the resources your VM needs are highly dependent on the workloads you deploy. If your VM does not have sufficient memory to support your workloads, it will fail to start.

You can customize the virtual machine's available resources using the `Deploy-Eflow` command's optional parameters. This is required to deploy EFLOW on a device with the minimum hardware requirements.

For example, the command below creates a virtual machine with 1 vCPU core, 1 GB of RAM (represented in MB), and 2 GB of disk space:

PowerShell

```
Deploy-Eflow -cpuCount 1 -memoryInMB 1024 -vmDataSize 2
```

For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

By default, the EFLOW Linux virtual machine has no DNS configuration. Deployments using DHCP will try to obtain the DNS configuration propagated by the DHCP server. Please check your DNS configuration to ensure internet connectivity. For more information, see [AzEFLW-DNS](#).

You can assign a GPU to your deployment to enable GPU-accelerated Linux modules. To gain access to these features, you will need to install the prerequisites detailed in [GPU acceleration for Azure IoT Edge for Linux on Windows](#).

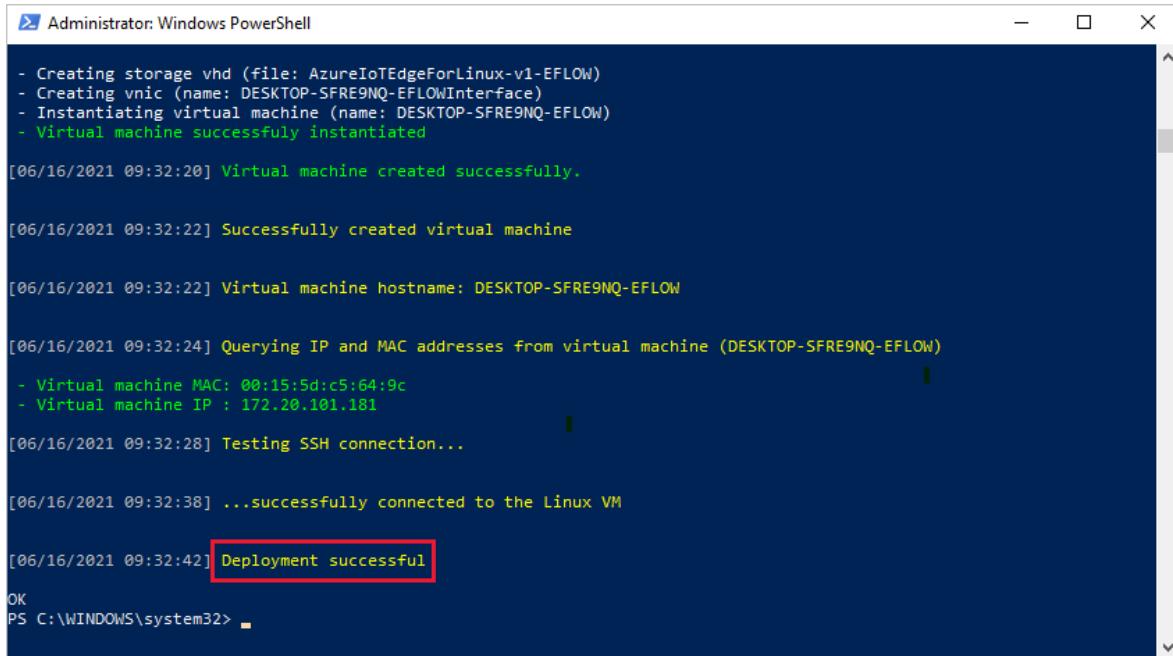
To use a GPU passthrough, add the `gpuName`, `gpuPassthroughType`, and `gpuCount` parameters to your `Deploy-Eflow` command. For information about all the optional parameters available, see [PowerShell functions for IoT Edge for Linux on Windows](#).

⚠️ Warning

Enabling hardware device passthrough may increase security risks. Microsoft recommends a device mitigation driver from your GPU's vendor, when applicable. For more information, see [Deploy graphics devices using discrete device assignment](#).

5. Enter 'Y' to accept the license terms.
6. Enter 'O' or 'R' to toggle **Optional diagnostic data** on or off, depending on your preference.

- Once the deployment is complete, the PowerShell window reports Deployment successful.



```
- Creating storage vhd (file: AzureIoTEdgeForLinux-v1-EFLOW)
- Creating vnic (name: DESKTOP-SFRE9NQ-EFLOWInterface)
- Instantiating virtual machine (name: DESKTOP-SFRE9NQ-EFLOW)
- Virtual machine successfully instantiated

[06/16/2021 09:32:20] Virtual machine created successfully.

[06/16/2021 09:32:22] Successfully created virtual machine

[06/16/2021 09:32:22] Virtual machine hostname: DESKTOP-SFRE9NQ-EFLOW

[06/16/2021 09:32:24] Querying IP and MAC addresses from virtual machine (DESKTOP-SFRE9NQ-EFLOW)

- Virtual machine MAC: 00:15:5d:c5:64:9c
- Virtual machine IP : 172.20.101.181

[06/16/2021 09:32:28] Testing SSH connection...

[06/16/2021 09:32:38] ...successfully connected to the Linux VM

[06/16/2021 09:32:42] Deployment successful
```

After a successful deployment, you are ready to provision your device.

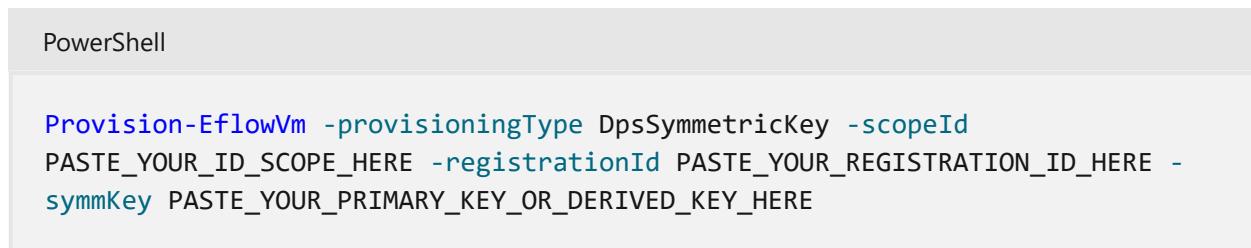
Provision the device with its cloud identity

Once the runtime is installed on your device, configure the device with the information it uses to connect to the device provisioning service and IoT Hub.

Have the following information ready:

- The DPS ID Scope value
- The device Registration ID you created
- Either the Primary Key from an individual enrollment, or a [derived key](#) for devices using a group enrollment.

Run the following command in an elevated PowerShell session with the placeholder values updated with your own values:



```
Provision-EflowVm -provisioningType DpsSymmetricKey -scopeId
PASTE_YOUR_ID_SCOPE_HERE -registrationId PASTE_YOUR_REGISTRATION_ID_HERE -
symmKey PASTE_YOUR_PRIMARY_KEY_OR_DERIVED_KEY_HERE
```

Verify successful installation

Verify that IoT Edge for Linux on Windows was successfully installed and configured on your IoT Edge device.

Individual enrollment

You can verify that the individual enrollment that you created in device provisioning service was used. Navigate to your device provisioning service instance in the Azure portal. Open the enrollment details for the individual enrollment that you created. Notice that the status of the enrollment is **assigned** and the device ID is listed.

1. Sign in to your IoT Edge for Linux on Windows virtual machine using the following command in your PowerShell session:

PowerShell

```
Connect-EflowVm
```

① Note

The only account allowed to SSH to the virtual machine is the user that created it.

2. Once you are logged in, you can check the list of running IoT Edge modules using the following Linux command:

Bash

```
sudo iotedge list
```

3. If you need to troubleshoot the IoT Edge service, use the following Linux commands.

- a. Retrieve the service logs.

Bash

```
sudo iotedge system logs
```

- b. Use the `check` tool to verify configuration and connection status of the device.

Bash

```
sudo iotedge check
```

ⓘ Note

On a newly provisioned device, you may see an error related to IoT Edge Hub:

- ✗ **production readiness: Edge Hub's storage directory is persisted on the host filesystem - Error**

Could not check current state of edgeHub container

This error is expected on a newly provisioned device because the IoT Edge Hub module isn't running. To resolve the error, in IoT Hub, set the modules for the device and create a deployment. Creating a deployment for the device starts the modules on the device including the IoT Edge Hub module.

When you create a new IoT Edge device, it displays the status code `417 -- The device's deployment configuration is not set` in the Azure portal. This status is normal, and means that the device is ready to receive a module deployment.

Uninstall IoT Edge for Linux on Windows

If you want to remove the Azure IoT Edge for Linux on Windows installation from your device, use the following commands.

1. Open Settings on Windows
2. Select Add or Remove Programs
3. Select *Azure IoT Edge* app
4. Select Uninstall

Next steps

The device provisioning service enrollment process lets you set the device ID and device twin tags at the same time as you provision the new device. You can use those values to target individual devices or groups of devices using automatic device management. Learn how to [Deploy and monitor IoT Edge modules at scale using the Azure portal](#) or [using Azure CLI](#).

You can also:

- Continue to [deploy IoT Edge modules](#) to learn how to deploy modules onto your device.
- Learn how to [manage certificates on your IoT Edge for Linux on Windows virtual machine](#) and transfer files from the host OS to your Linux virtual machine.
- Learn how to [configure your IoT Edge devices to communicate through a proxy server](#).

dTPM access for Azure IoT Edge for Linux on Windows

Article • 05/29/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

A Trusted platform module (TPM) chip is a secure crypto-processor that is designed to carry out cryptographic operations. This technology is designed to provide hardware-based, security-related functions. The Azure IoT Edge for Linux on Windows (EFLW) virtual machine doesn't have a virtual TPMs attached to the VM. However, the user can enable or disable the TPM passthrough feature, that allows the EFLW virtual machine to use the Windows host OS TPM. The TPM passthrough feature enables two main scenarios:

- Use TPM technology for IoT Edge device provisioning using Device Provisioning Service (DPS)
- Read-only access to cryptographic keys stored inside the TPM.

This article describes how to develop a sample code in C# to read cryptographic keys stored inside the device TPM.

Important

The access to the TPM keys is limited to read-only. If you want to write keys to the TPM, you need to do it from the Windows host OS.

Prerequisites

- A Windows host OS with a TPM or vTPM (if using Windows host OS virtual machine).
- EFLW virtual machine with TPM passthrough enabled. Using an elevated PowerShell session, use `Set-EflowVmFeature -feature "DpsTpm" -enable` to enable

TPM passthrough. For more information, see [Set-EflowVmFeature to enable TPM passthrough](#).

- Ensure that the NV index (default index=3001) is initialized with 8 bytes of data. The default AuthValue used by the sample is {1,2,3,4,5,6,7,8} which corresponds to the NV (Windows) Sample in the TSS.MSR libraries when writing to the TPM. All index initialization must take place on the Windows Host before reading from the EFLOW VM. For more information about TPM samples, see [TSS.MSR](#).

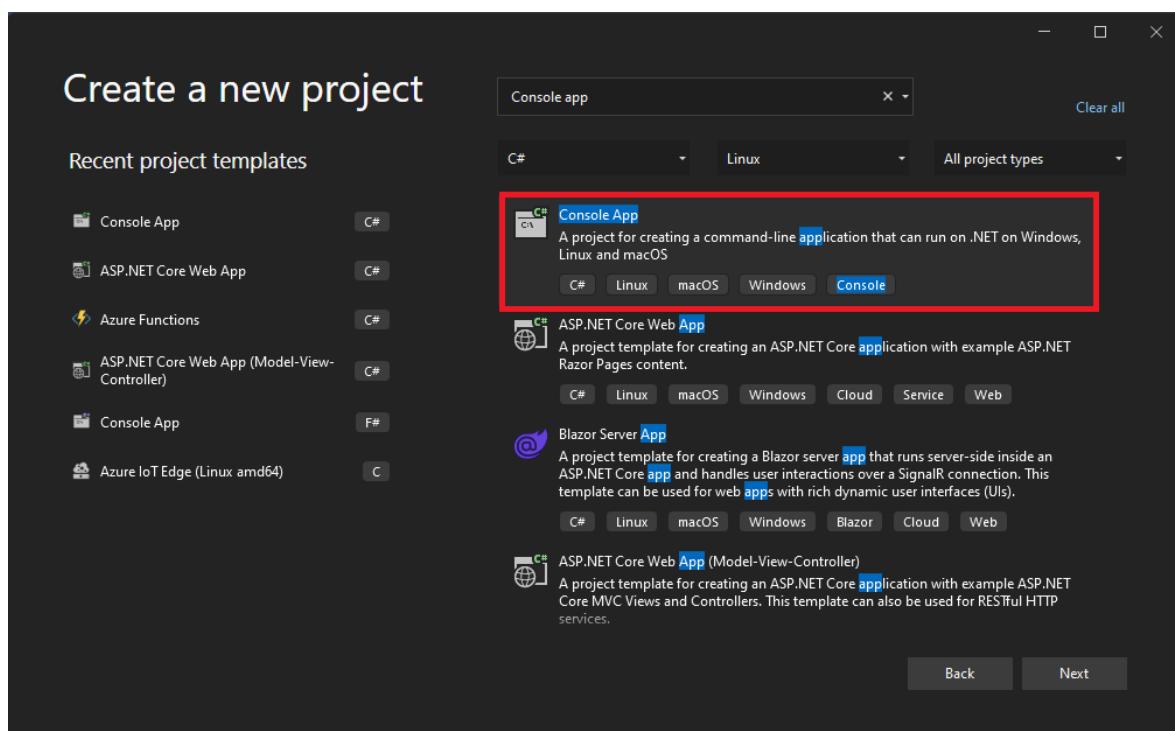
⚠ Warning

Enabling TPM passthrough to the virtual machine may increase security risks.

Create the dTPM executable

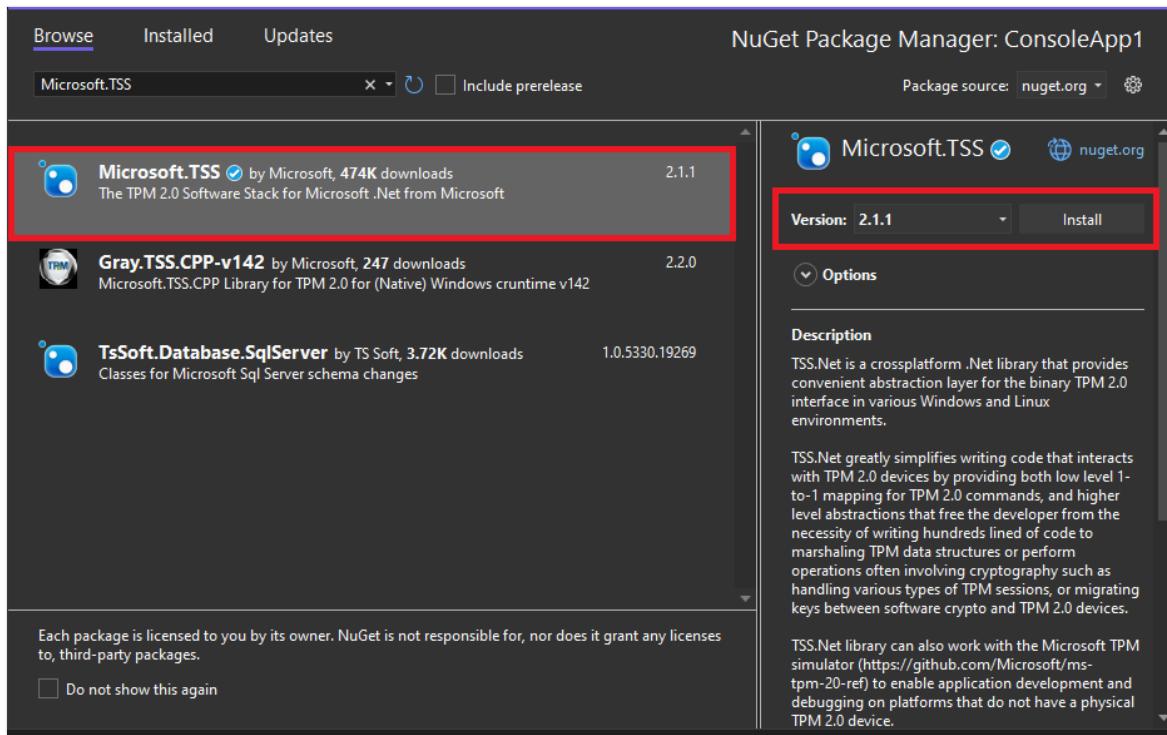
The following steps show you how to create a sample executable to access a TPM index from the EFLOW VM. For more information about EFLOW TPM passthrough, see [Azure IoT Edge for Linux on Windows Security](#).

1. Open Visual Studio 2019 or 2022.
2. Select **Create a new project**.
3. Choose **Console App** in the list of templates then select **Next**.

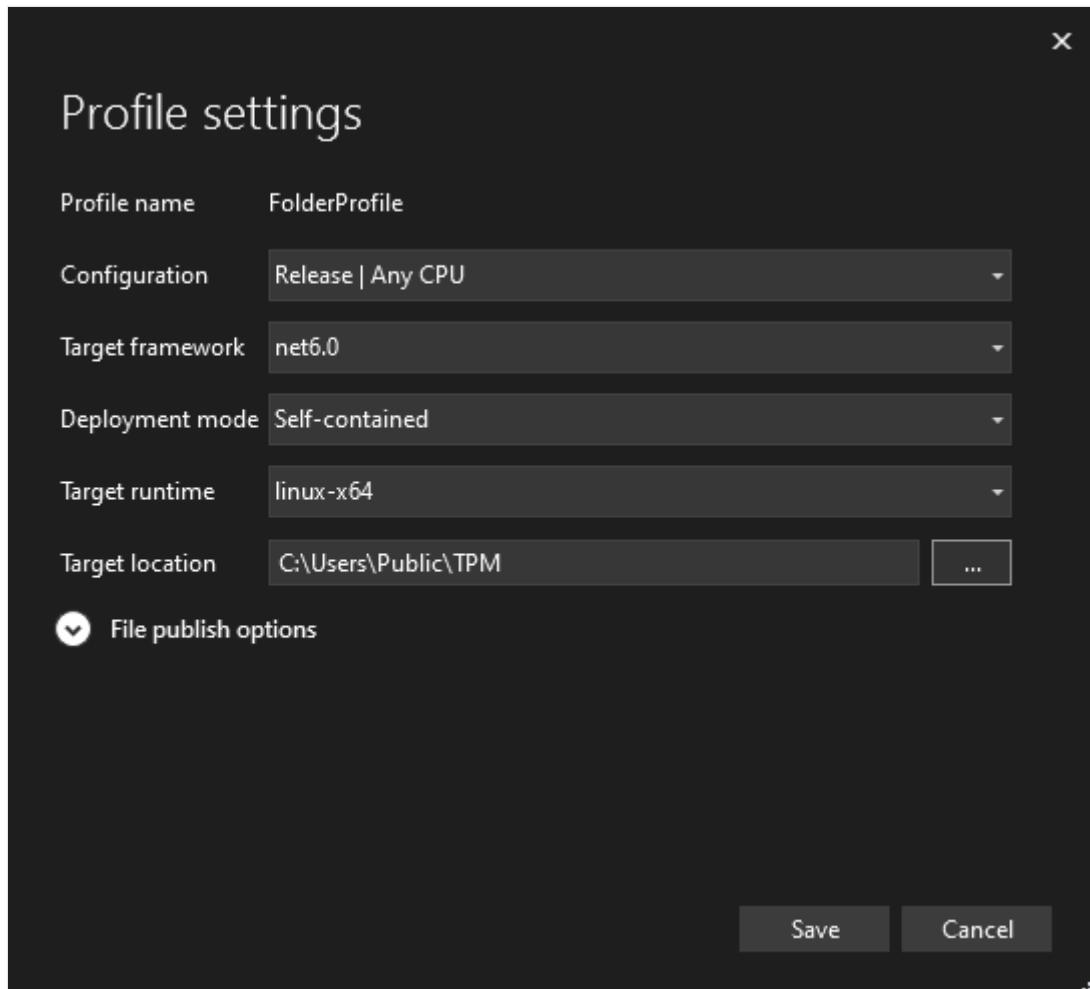


4. Fill in the **Project Name**, **Location** and **Solution Name** fields then select **Next**.

5. Choose a target framework. The latest .NET 6.0 LTS version is preferred. After choosing a target framework, select **Create**. Visual Studio creates a new console app solution.
6. In **Solution Explorer**, right-click the project name and select **Manage NuGet Packages**.
7. Select **Browse** and then search for `Microsoft.TSS`. For more information about this package, see [Microsoft.TSS](#).
8. Choose the **Microsoft.TSS** package from the list then select **Install**.



9. Edit the `Program.cs` file and replace the contents with the [EFLOW TPM sample code - Program.cs](#).
10. Select **Build > Build solution** to build the project. Verify the build is successful.
11. In **Solution Explorer**, right-click the project then select **Publish**.
12. In the **Publish** wizard, choose **Folder > Folder**. Select **Browse** and choose an output location for the executable file to be generated. Select **Finish**. After the publish profile is created, select **Close**.
13. On the **Publish** tab, select **Show all settings** link. Change the following configurations then select **Save**.
 - Target Runtime: `linux-x64`.
 - Deployment mode: `Self-contained`.



14. Select **Publish** then wait for the executable to be created.

If publish succeeds, you should see the new files created in your output folder.

Copy and run the executable

Once the executable file and dependency files are created, you need to copy the folder to the EFLOW virtual machine. The following steps show you how to copy all the necessary files and how to run the executable inside the EFLOW virtual machine.

1. Start an elevated *PowerShell* session using **Run as Administrator**.
2. Change directory to the parent folder that contains the published files. For example, if your published files are under the folder *TPM* in the directory `C:\Users\User`. You can use the following command to change to the parent folder.

```
PowerShell  
cd "C:\Users\User"
```

3. Create a *tar* file with all the files created in previous steps. For example, if you have all your files under the folder *TPM*, you can use the following command to create the *TPM.tar* file.

```
PowerShell
```

```
tar -cvzf TPM.tar ".\TPM"
```

4. Once the *TPM.tar* file is created successfully, use the `Copy-EflowVmFile` cmdlet to copy the *tar* file created to the EFLOW VM. For example, if you have the *tar* file name *TPM.tar* in the directory `C:\Users\User`. you can use the following command to copy to the EFLOW VM.

```
PowerShell
```

```
Copy-EflowVmFile -fromFile "C:\Users\User\TPM.tar" -toFile  
"/home/iotedge-user/" -pushFile
```

5. Connect to the EFLOW virtual machine.

```
PowerShell
```

```
Connect-EflowVm
```

6. Change directory to the folder where you copied the *tar* file and check the file is available. If you used the example above, when connected to the EFLOW VM, you'll already be at the *iotedge-user* root folder. Run the `ls` command to list the files and folders.

7. Run the following command to extract all the content from the *tar* file.

```
Bash
```

```
tar -xvzf TPM.tar
```

8. After extraction, you should see a new folder with all the TPM files.

9. Change directory to the *TPM* folder.

```
Bash
```

```
cd TPM
```

10. Add executable permission to the main executable file. For example, if your project name was *TPMRead*, your main executable is named *TPMRead*. Run the following command to make it executable.

```
Bash
```

```
chmod +x TPMRead
```

11. To solve an [ICU globalization issue](#), run the following command. For example, if your project name is *TPMTest* run:

```
Bash
```

```
sed -i '\"configProperties\": /a \\\t\"System.Globalization.Invariant\": true,' TPMTest.runtimeconfig.json
```

12. The last step is to run the executable file. For example, if your project name is *TPMTest*, run the following command:

```
Bash
```

```
./TPMTest
```

You should see an output similar to the following.

```
iotedge-user@DESKTOP-FCABRER-EFLOW [ ~/TPM ]$ ./TPMTest
AbrmdWrapper: Got pointer to TctiProvInfo from abrmd!
AbrmdWrapper: Unmarshaled TctiProvInfo
AbrmdWrapper: Initial call to tcti_init_fn() returned 0; ctxSize = 128
AbrmdWrapper: Successfully initialized TCTI ctx
AbrmdWrapper: Unmarshaled TCTI_CTX
Closing TCTI conn
TCTI conn closed!
AbrmdWrapper: Got pointer to TctiProvInfo from abrmd!
AbrmdWrapper: Unmarshaled TctiProvInfo
AbrmdWrapper: Initial call to tcti_init_fn() returned 0; ctxSize = 128
AbrmdWrapper: Successfully initialized TCTI ctx
AbrmdWrapper: Unmarshaled TCTI_CTX
Reading NVIndex 3001.
Read Bytes: 00-01-02-03-04-05-06-07
Closing TCTI conn
TCTI conn closed!
Press Any Key to continue.
```

Next steps

- Learn How to develop IoT Edge modules with Linux containers using IoT Edge for Linux on Windows.

Nested virtualization for Azure IoT Edge for Linux on Windows

Article • 06/06/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

There are three forms of nested virtualization compatible with Azure IoT Edge for Linux on Windows. Users can choose to deploy through a local virtual machine (using Hyper-V hypervisor), VMware Windows virtual machine, or Azure Virtual Machine. This article provides clarity on which option is best for their scenario and provide insight into configuration requirements.

Note

Ensure to enable one [networking option](#) for nested virtualization. Failing to do so will result in EFLOW installation errors.

Deployment on local VM

This is the baseline approach for any Windows VM that hosts Azure IoT Edge for Linux on Windows. For this case, nested virtualization needs to be enabled before starting the deployment. Read [Run Hyper-V in a Virtual Machine with Nested Virtualization](#) for more information on how to configure this scenario.

If you're using Windows Server or Azure Stack HCI, make sure you [install the Hyper-V role](#).

Deployment on Windows VM on VMware ESXi

Intel-based VMware ESXi [6.7](#) and [7.0](#) versions can host Azure IoT Edge for Linux on Windows on top of a Windows virtual machine. Read [VMware KB2009916](#) for more information on VMware ESXi nested virtualization support.

To set up an Azure IoT Edge for Linux on Windows on a VMware ESXi Windows virtual machine, use the following steps:

1. Create a Windows virtual machine on the VMware ESXi host. For more information about VMware VM deployment, see [VMware - Deploying Virtual Machines](#).

 **Note**

If you're creating a Windows 11 virtual machine, ensure to meet the minimum requirements by Microsoft to run Windows 11. For more information about Windows 11 VM VMware support, see [Installing Windows 11 as a guest OS on VMware](#).

1. Turn off the virtual machine created in previous step.
2. Select the Windows virtual machine and then **Edit settings**.
3. Search for *Hardware virtualization* and turn on *Expose hardware assisted virtualization to the guest OS*.
4. Select **Save** and start the virtual machine.
5. Install Hyper-V hypervisor. If you're using Windows client, make sure you [Install Hyper-V on Windows 10](#). If you're using Windows Server, make sure you [install the Hyper-V role](#).

 **Note**

For VMware Windows virtual machines, if you plan to use an **external virtual switch** for the EFLOW virtual machine networking, make sure you enable *Promiscuous mode*. For more information, see [Configuring promiscuous mode on a virtual switch or portgroup](#). Failing to do so will result in EFLOW installation errors.

Deployment on Azure VMs

Azure IoT Edge for Linux on Windows isn't compatible on an Azure VM running the Server SKU unless a script is executed that brings up a default switch. For more information on how to bring up a default switch, see [Create virtual switch for Linux on Windows](#).

 **Note**

Any Azure VMs that is supposed to host EFLOW must be a VM that supports nested virtualization. Also, Azure VMs do not support using an **external virtual**

switch.

How to connect a USB device to Azure IoT Edge for Linux on Windows

Article • 05/31/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

In some scenarios, your workloads need to get data or communicate with USB devices. Because Azure IoT Edge for Linux on Windows (EFLOW) runs as a virtual machine, you need to connect these devices to the virtual machine. This article guides you through the steps necessary to connect a USB device to the EFLOW virtual machine using the USB/IP open-source project named [usbipd-win](#).

Setting up the USB/IP project on your Windows machine enables common developer USB scenarios like flashing an Arduino, connecting a USB serial device, or accessing a smartcard reader directly from the EFLOW virtual machine.

Warning

USB over IP provides a generic mechanism for redirecting USB devices using the network between the Windows host OS and the EFLOW virtual machine. Some devices that are sensitive to network latency might experience issues. Additionally, some devices might not function as expected due to driver compatibility issues. Ensure that your devices work as expected before deploying to production. For more information about USB/IP tested devices, see [USBIP-Win - Wiki - Tested Devices](#).

Prerequisites

- Azure IoT Edge for Linux on Windows 1.3.1 update or higher. For more information about EFLOW release notes, see [EFLOW Releases](#).
- A machine with an x64/x86 processor is required, *usbipd-win* doesn't support ARM64.

Note

To check your Azure IoT Edge for Linux on Windows version, go to *Add or Remove Programs* and then search for *Azure IoT Edge*. The installed version is listed under *Azure IoT Edge*. If you need to update to the latest version, see [Azure IoT Edge for Linux on Windows updates](#).

Install the UsbIp-Win project

Support for connecting USB devices isn't natively available with EFLOW. You'll need to install the open-source [usbipd-win](#) project using the following steps:

1. Go to the [latest release page for the usbipd-win](#) project.
2. Choose and download the *usbipd-win_x.y.z.msi* file. (You may get a warning asking you to confirm that you trust the downloaded installer).
3. Run the downloaded *usbipd-win_x.y.z.msi* installer file.

Note

Alternatively, you can also install the usbipd-win project using [Windows Package Manager](#) (`winget`). If you have already installed `winget`, use the command: `winget install --interactive --exact dorsel.usbipd-win` to install usbipd-win. If you don't use the `--interactive` parameter, `winget` may immediately restart your computer if needed to install the drivers.

The UsbIp-Win installs:

- A service called `usbipd` (USBIP Device Host). You can check the status of this service using the *Services* app in Windows.
- A command line tool `usbipd`. The location of this tool is added to the PATH environment variable.
- A firewall rule called `usbipd` to allow all local subnets to connect to the service. You can modify this firewall rule to fine tune access control.

At this point, a service is running on Windows to share USB devices, and the necessary tools are installed in the EFLOW virtual machine to attach to shared devices.

Warning

If you have an open PowerShell session, make sure to close it and open a new one to load the `usbipd` command line tool.

Attach a USB device to the EFLOW VM

The following steps provide a sample EFLOW PowerShell cmdlet to attach a USB device to the EFLOW VM. If you want to manually execute the needed commands, see [How to use usbip-win](#).

ⓘ Important

The following functions are samples that are not meant to be used in production deployments. For production use, ensure you validate the functionality and create your own functions based on these samples. The sample functions are subject to change and deletion.

1. Go to [EFLOW-Util](#) and download the EFLOW-USBIP sample PowerShell module.
2. Open an elevated PowerShell session by starting with **Run as Administrator**.
3. Import the downloaded EFLOW-USBIP module.

```
PowerShell  
Import-Module "<path-to-module>/EflowUtil-Usbip.psm1"
```

4. List all of the USB devices connected to Windows.

```
PowerShell  
Get-EflowUSBDevices
```

5. List all the network interfaces and get the Windows host OS IP address

```
PowerShell  
ipconfig
```

6. Select the *bus ID* of the device you'd like to attach to the EFLOW.

```
PowerShell
```

```
Add-EflowUSBDevices -busid <busid> -hostIp <host-ip>
```

7. Check the device was correctly attached to the EFLOW VM.

```
PowerShell
```

```
Invoke-EflowVmCommand "lsusb"
```

8. Once you're finished using the device in EFLOW, you can either physically disconnect the USB device or run this command from an elevated PowerShell session.

```
PowerShell
```

```
Remove-EflowUSBDevices -busid <busid>
```

Important

The attachment from the EFLOW VM to the USB device does not persist across reboots. To attach the USB device after reboot, you may need to create a bash script that runs during startup and connects the device using the `usbip` bash command. For more information about how to attach the device on the EFLOW VM side, see [Add-EflowUSBDevices](#).

To learn more about how USB over IP, see the [Connecting USB devices to WSL](#) and the [usbipd-win repo on GitHub](#).

Next steps

Follow the steps in [How to develop IoT Edge modules with Linux containers using IoT Edge for Linux on Windows](#). to develop and debug a module with IoT Edge for Linux on Windows.

GPU acceleration for Azure IoT Edge for Linux on Windows

Article • 05/29/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

GPUs are a popular choice for artificial intelligence computations, because they offer parallel processing capabilities and can often execute vision-based inferencing faster than CPUs. To better support artificial intelligence and machine learning applications, Azure IoT Edge for Linux on Windows (EFLLOW) can expose a GPU to the virtual machine's Linux module.

Azure IoT Edge for Linux on Windows supports several GPU passthrough technologies, including:

- **Direct Device Assignment (DDA)** - GPU cores are allocated either to the Linux virtual machine or the host.
- **GPU-Paravirtualization (GPU-PV)** - The GPU is shared between the Linux virtual machine and the host.

You must select the appropriate passthrough method during deployment to match the supported capabilities of your device's GPU hardware.

Important

These features may include components developed and owned by NVIDIA Corporation or its licensors. The use of the components is governed by the NVIDIA End-User License Agreement located [on NVIDIA's website](#) ↗.

By using GPU acceleration features, you are accepting and agreeing to the terms of the NVIDIA End-User License Agreement.

Prerequisites

The GPU acceleration features of Azure IoT Edge for Linux on Windows currently supports a select set of GPU hardware. Additionally, use of this feature may require specific versions of Windows.

The supported GPUs and required Windows versions are listed below:

[+] Expand table

| Supported GPUs | GPU Passthrough Type | Supported Windows Versions |
|-----------------------------|----------------------|---|
| NVIDIA T4, A2 | DDA | Windows Server 2019
Windows Server 2022
Windows 10/11 (Pro, Enterprise, IoT Enterprise) |
| NVIDIA GeForce, Quadro, RTX | GPU-PV | Windows 10/11 (Pro, Enterprise, IoT Enterprise) |
| Intel iGPU | GPU-PV | Windows 10/11 (Pro, Enterprise, IoT Enterprise) |

ⓘ Important

GPU-PV support may be limited to certain generations of processors or GPU architectures as determined by the GPU vendor. For more information, see [Intel's iGPU driver documentation](#) or [NVIDIA's CUDA for WSL Documentation](#).

Windows Server 2019 users must use minimum build 17763 with all current cumulative updates installed.

Windows 10 users must use the [November 2021 update](#) build 19044.1620 or higher. After installation, you can verify your build version by running `winver` at the command prompt.

GPU passthrough is not supported with nested virtualization, such as running EFLOW in a Windows virtual machine.

System setup and installation

The following sections contain setup and installation information, according to your GPU.

NVIDIA T4/A2 GPUs

For T4/A2 GPUs, Microsoft recommends installing a device mitigation driver from your GPU's vendor. While optional, installing a mitigation driver may improve the security of your deployment. For more information, see [Deploy graphics devices using direct device assignment](#).

Warning

Enabling hardware device passthrough may increase security risks. Microsoft recommends a device mitigation driver from your GPU's vendor, when applicable.

For more information, see [Deploy graphics devices using discrete device assignment](#).

NVIDIA GeForce/Quadro/RTX GPUs

For NVIDIA GeForce/Quadro/RTX GPUs, download and install the [NVIDIA CUDA-enabled driver for Windows Subsystem for Linux \(WSL\)](#) to use with your existing CUDA ML workflows. Originally developed for WSL, the CUDA for WSL drivers are also used for Azure IoT Edge for Linux on Windows.

Windows 10 users must also [install WSL](#) because some of the libraries are shared between WSL and Azure IoT Edge for Linux on Windows.

Intel iGPUs

For Intel iGPUs, download and install the [Intel Graphics Driver with WSL GPU support](#).

Windows 10 users must also [install WSL](#) because some of the libraries are shared between WSL and Azure IoT Edge for Linux on Windows.

Enable GPU acceleration in your Azure IoT Edge Linux on Windows deployment

Once system setup is complete, you are ready to [create your deployment of Azure IoT Edge for Linux on Windows](#). During this process, you must [enable GPU](#) as part of EFLOW deployment.

For example, the following commands create a GPU-enabled virtual machine with either an NVIDIA A2 GPU or Intel Iris Xe Graphics card.

PowerShell

```
#Deploys EFLOW with NVIDIA A2 assigned to the EFLOW VM  
Deploy-Eflow -gpuPassthroughType DirectDeviceAssignment -gpuCount 1 -gpuName  
"NVIDIA A2"  
  
#Deploys EFLOW with Intel(R) Iris(R) Xe Graphics assigned to the EFLOW VM  
Deploy-Eflow -gpuPassthroughType ParaVirtualization -gpuCount 1 -gpuName  
"Intel(R) Iris(R) Xe Graphics"
```

To find the name of your GPU, you can run the following command or look for Display adapters in Device Manager.

PowerShell

```
(Get-WmiObject win32_VideoController).caption
```

Once installation is complete, you are ready to deploy and run GPU-accelerated Linux modules through Azure IoT Edge for Linux on Windows.

Configure GPU acceleration in an existing Azure IoT Edge Linux on Windows deployment

Assigning the GPU at deployment time will result in the most straightforward experience. However, to enable or disable the GPU after deployment use the 'set-eflowvm' command. When using 'set-eflowvm' the default parameter will be used for any argument not specified. For example,

PowerShell

```
#Deploys EFLOW without a GPU assigned to the EFLOW VM  
Deploy-Eflow -cpuCount 4 -memoryInMB 16384  
  
#Assigns NVIDIA A2 GPU to the existing deployment (cpu and memory must still  
be specified, otherwise they will be set to the default values)  
Set-EflowVM -cpuCount 4 -memoryInMB 16384 -gpuName "NVIDIA A2" -  
gpuPassthroughType DirectDeviceAssignment -gpuCount 1  
  
#Reduces the cpuCount and memory (GPU must still be specified, otherwise the  
GPU will be removed)  
Set-EflowVM -cpuCount 2 -memoryInMB 4096 -gpuName "NVIDIA A2" -  
gpuPassthroughType DirectDeviceAssignment -gpuCount 1  
  
#Removes NVIDIA A2 GPU from the existing deployment  
Set-EflowVM -cpuCount 2 -memoryInMB 4096
```

Next steps

Get Started with Samples

Visit our [EFLow Samples Page](#) to discover several GPU samples which you can try and use. These samples illustrate common manufacturing and retail scenarios such as defect detection, worker safety, and inventory management. These open-source samples can serve as a solution template for building your own vision-based machine learning application.

Learn More from our Partners

Several GPU vendors have provided user guides on getting the most of their hardware and software with EFLow.

- Learn how to run Intel OpenVINO™ applications on EFLow by following [Intel's guide on iGPU with Azure IoT Edge for Linux on Windows \(EFLow\) & OpenVINO™ Toolkit](#) and [reference implementations](#).
- Get started with deploying CUDA-accelerated applications on EFLow by following [NVIDIA's EFLow User Guide for GeForce/Quadro/RTX GPUs](#).

Note

This guide does not cover DDA-based GPUs such as NVIDIA T4 or A2.

Dive into the Technology

Learn more about GPU passthrough technologies by visiting the [DDA documentation](#) and [GPU-PV blog post](#).

Networking configuration for Azure IoT Edge for Linux on Windows

Article • 05/31/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article helps you decide which networking option is best for your scenario and provide insights into IoT Edge for Linux on Windows (EFLOW) configuration requirements.

To connect the IoT Edge for Linux on Windows (EFLOW) virtual machine over a network to your host, to other virtual machines on your Windows host, and to other devices/locations on an external network, the virtual machine networking must be configured accordingly.

The easiest way to establish basic networking on Windows Client SKUs is by using the **default switch**, which is already created when enabling the Windows Hyper-V feature. However, on Windows Server SKUs devices, networking it's a bit more complicated as there's no **default switch** available. For more information about virtual switch creation for Windows Server, see [Create virtual switch for Linux on Windows](#).

For more information about EFLOW networking concepts, see [IoT Edge for Linux on Windows networking](#).

Configure VM virtual switch

The first step before deploying the EFLOW virtual machine is to determine which type of virtual switch you use. For more information about EFLOW supported virtual switches, see [EFLOW virtual switch choices](#). Once you determine the type of virtual switch that you want to use, make sure to create the virtual switch correctly. For more information about virtual switch creation, see [Create a virtual switch for Hyper-V virtual machines](#).

Note

If you're using Windows client and you want to use the **default switch**, then no switch creation is needed and no `-vSwitchType` and `-vSwitchName` parameters are needed.

ⓘ Note

If you're using a Windows virtual machine inside VMware infrastructure and **external switch**, please see [EFLOW nested virtualization](#).

After creating the virtual switch and before starting your deployment, make sure that your virtual switch name and type is correctly set up and is listed under the Windows host OS. To list all the virtual switches in your Windows host OS, in an elevated PowerShell session, use the following PowerShell cmdlet:

PowerShell

```
Get-VmSwitch
```

Depending on the virtual switches of the Windows host, the output should be similar to the following:

Output

| Name | SwitchType | NetAdapterInterfaceDescription |
|----------------|------------|--------------------------------|
| Default Switch | Internal | |
| IntOff | Internal | |
| EFLOW-Ext | External | |

To use a specific virtual switch(**internal** or **external**), make sure you specify the correct parameters: `-vSwitchName` and `vSwitchType`. For example, if you want to deploy the EFLOW VM with an **external switch** named **EFLOW-Ext**, then in an elevated PowerShell session use the following command:

PowerShell

```
Deploy-Eflow -vSwitchType "External" -vSwitchName "EFLOW-Ext"
```

Configure VM IP address allocation

The second step after deciding the type of virtual switch you're using is to determine the type of IP address allocation of the virtual switch. For more information about IP allocation options, see [Eflow supported IP allocations](#). Depending on the type of virtual switch you're using, make sure to use a supported IP address allocation mechanism.

By default, if no **static IP** address is set up, the Eflow VM tries to allocate an IP address to the virtual switch using **DHCP**. Make sure that there's a DHCP server on the virtual switch network; if not available, the Eflow VM installation fails to allocate an IP address and installation fails. If you're using the **default switch**, then there's no need to check for a DHCP server, as the virtual switch already has DHCP by default. However, if using an **internal** or **external** virtual switch, you can check using the following steps:

1. Open a command prompt.
2. Display all the IP configurations and information

```
Windows Command Prompt
```

```
ipconfig /all
```

3. If you're using an **external** virtual switch, check the network interface used for creating the virtual switch. If you're using an **internal** virtual switch, just look for the name used for the switch. Once the switch is located, check if **DHCP Enabled** says **Yes** or **No**, and check the **DHCP server** address.

If you're using a **static IP**, you have to specify three parameters during Eflow deployment: **-ip4Address**, **ip4GatewayAddress** and **ip4PrefixLength**. If one parameter is missing or incorrect, the Eflow VM installation fails to allocate an IP address and installation fails. For more information about Eflow VM deployment, see [PowerShell functions for IoT Edge for Linux on Windows](#). For example, if you want to deploy the Eflow VM with an **external** switch named **Eflow-Ext**, and a static IP configuration, with an IP address equal to **192.168.0.2**, gateway IP address equal to **192.168.0.1** and IP prefix length equal to **24**, then in an elevated PowerShell session use the following command:

```
PowerShell
```

```
Deploy-Eflow -vSwitchType "External" -vSwitchName "Eflow-Ext" -ip4Address  
"192.168.0.2" -ip4GatewayAddress "192.168.0.1" -ip4PrefixLength "24"
```

 **Tip**

The EFLOW VM will keep the same MAC address for the main (used during deployment) virtual switch across reboots. If you are using DHCP MAC address reservation, you can get the main virtual switch MAC address using the PowerShell cmdlet: `Get-EflowVmAddr`.

Check IP allocation

There are multiple ways to check the IP address that was allocated to the EFLOW VM. First, using an elevated PowerShell session, use the EFLOW cmdlet:

Bash

```
Get-EflowVmAddr
```

The output should be something similar to the following:

Output

```
C:\> Get-EflowVmAddr

[03/31/2022 12:54:31] Querying IP and MAC addresses from virtual machine
(DESKTOP-EFLOW)

- Virtual machine MAC: 00:15:5d:4e:15:2c
- Virtual machine IP : 172.27.120.111 retrieved directly from virtual
machine
00:15:5d:4e:15:2c
172.27.120.111
```

Another way, is using the `Connect-Eflow` cmdlet to remote into the VM, and then you can use the `ifconfig eth0` bash command, and check for the *eth0* interface. The output should be similar to the following:

Output

```
eth0      Link encap:Ethernet  HWaddr 00:15:5d:4e:15:2c
          inet  addr:172.27.120.111  Bcast:172.27.127.255  Mask:255.255.240.0
          inet6 addr: fe80::215:5dff:fe4e:152c/64 Scope:Link
            UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
            RX packets:5636 errors:0 dropped:0 overruns:0 frame:0
            TX packets:2214 errors:0 dropped:0 overruns:0 carrier:0
            collisions:0 txqueuelen:1000
            RX bytes:766832 (766.8 KB)  TX bytes:427274 (427.2 KB)
```

Configure VM DNS servers

By default, the EFLOW virtual machine has no DNS configuration. Deployments using DHCP tries to obtain the DNS configuration propagated by the DHCP server. If you're using a **static IP**, the DNS server needs to be set up manually. For more information about EFLOW VM DNS, see [EFLOW DNS configuration](#).

To check the DNS servers used by the default interface (*eth0*), you can use the following command:

Bash

```
resolvectl | grep eth0 -A 8
```

The output should be something similar to the following. Check the IP addresses of the "Current DNS Servers" and "DNS Servers" fields of the list. If there's no IP address, or the IP address isn't a valid DNS server IP address, then the DNS service won't work.

Output

```
Link 2 (eth0)
  Current Scopes: DNS
  LLMNR setting: yes
  MulticastDNS setting: no
  DNSOverTLS setting: no
  DNSSEC setting: no
  DNSSEC supported: no
  Current DNS Server: 172.27.112.1
  DNS Servers: 172.27.112.1
```

If you need to manually set up the DNS server addresses, you can use the EFLOW PowerShell cmdlet `Set-EflowVmDNSServers`. For more information about EFLOW VM DNS configuration, see [PowerShell functions for IoT Edge for Linux on Windows](#).

Check DNS resolution

There are multiple ways to check the DNS resolution.

First, from inside the EFLOW VM, use the `resolvectl query` command to query a specific URL. For example, to check if the name resolution is working for the address *microsoft.com*, use:

Bash

```
resolvectl query microsoft.com
```

The output should be similar to the following:

Output

```
PS C:\> resolvectl query
microsoft.com: 40.112.72.205
               40.113.200.201
               13.77.161.179
               104.215.148.63
               40.76.4.15

-- Information acquired via protocol DNS in 1.9ms.
-- Data is authenticated: no
```

You can also use the `dig` command to query a specific URL. For example, to check if the name resolution is working for the address *microsoft.com*, use:

Bash

```
dig microsoft.com
```

The output should be similar to the following:

Output

```
PS C:\> dig microsoft.com
; <>> DiG 9.16.22 <>> microsoft.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36427
;; flags: qr rd ra; QUERY: 1, ANSWER: 5, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;microsoft.com.           IN      A

;; ANSWER SECTION:
microsoft.com.      0      IN      A      40.112.72.205
microsoft.com.      0      IN      A      40.113.200.201
microsoft.com.      0      IN      A      13.77.161.179
microsoft.com.      0      IN      A      104.215.148.63
microsoft.com.      0      IN      A      40.76.4.15

;; Query time: 11 msec
;; SERVER: 127.0
```

Next steps

Read more about [Azure IoT Edge for Linux on Windows Security](#).

Stay up-to-date with the latest [IoT Edge for Linux on Windows updates](#).

Azure IoT Edge for Linux on Windows virtual switch creation

Article • 03/27/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Azure IoT Edge for Linux on Windows uses a virtual switch on the host machine to communicate with the virtual machine. Windows desktop versions come with a default switch that can be used, but Windows Server *doesn't*. Before you can deploy IoT Edge for Linux on Windows to a Windows Server device, you need to create a virtual switch. Furthermore, you can use this guide to create your custom virtual switch, if needed.

This article shows you how to create a virtual switch on a Windows device to install IoT Edge for Linux on Windows. This process is divided into the following steps:

- Create a virtual switch
- Create a NAT table
- Install and set up a DHCP server

Prerequisites

- A Windows device. For more information on supported Windows versions, see [Operating Systems](#).
- Hyper-V role installed on the Windows device. For more information on how to enable Hyper-V, see [Install and provision Azure IoT Edge for Linux on a Windows device](#).

Create virtual switch

The following steps in this section are a generic guide for a virtual switch creation. Ensure that the virtual switch configuration aligns with your networking environment.

Note

The following steps describe how to create an **Internal** or **Private** virtual switch. For more information on creating an **External** switch instead, see [Create a virtual switch for Hyper-V virtual machines](#). Note that if you're using an Azure VM, the virtual switch can't be **External**.

1. Open PowerShell in an elevated session. You can do so by opening the **Start** pane on Windows and typing in "PowerShell". Right-click the **Windows PowerShell** app that shows up and select **Run as administrator**.
2. Check the virtual switches on the Windows host and make sure you don't already have a virtual switch that can be used. You can do so by running the following [Get-VMSwitch](#) command in PowerShell:

```
PowerShell
```

```
Get-VMSwitch
```

If a virtual switch named **Default Switch** is already created and you don't need a custom virtual switch, you should be able to install IoT Edge for Linux on Windows without following the rest of the steps in this guide.

3. Create a new VM switch with a name of your choice and an **Internal** or **Private** switch type by running the following [New-VMSwitch](#) command, replacing the placeholder values:

```
PowerShell
```

```
New-VMSwitch -Name "{switchName}" -SwitchType {switchType}
```

4. To get the IP address for the switch you created, you must first get its interface index. You can get this value by running the following [Get-NetAdapter](#) command, replacing the placeholder value:

```
PowerShell
```

```
(Get-NetAdapter -Name "{switchName}").ifIndex
```

You may need to change the value for the `Name` parameter to follow the `vEthernet ({switchName})` template if you receive an error when you try to run this command. You should receive similar output to the following example:

```

PS C:\Users\contoso> New-VMSwitch -Name "Default Switch" -SwitchType Internal
Name      SwitchType NetAdapterInterfaceDescription
----      ----- -----
Default Switch Internal

PS C:\Users\contoso> (Get-NetAdapter [-Name "Default Switch"]).ifIndex
Get-NetAdapter : No MSFT_NetAdapter objects found with property 'Name' equal to 'Default Switch'. Verify the value of
the property and retry.
At line:1 char:2
+ (Get-NetAdapter -Name "Default Switch").ifIndex
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Default Switch:String) [Get-NetAdapter], CimJobException
+ FullyQualifiedErrorId : CmdletizationQuery_NotFound_Name,Get-NetAdapter
PS C:\Users\contoso> (Get-NetAdapter [-Name "vEthernet (Default Switch)"]).ifIndex
19

```



Take note of the interface index value, as you'll need to use it in future steps.

5. The resulting virtual switch IP address will be different for each environment. Note that for the rest of the commands in this guide you will make use of IP addresses that are derived from the 172.20.X.Y family. However, you can you use your own address family and IP addresses.

You'll create and use the following IP addresses:

[\[+\] Expand table](#)

| IP address | Template | Example |
|------------|-----------------|--------------|
| Gateway IP | xxx.xxx.xxx.1 | 172.20.0.1 |
| NAT IP | xxx.xxx.xxx.0 | 172.20.0.0 |
| Start IP | xxx.xxx.xxx.100 | 172.20.0.100 |
| End IP | xxx.xxx.xxx.200 | 172.20.0.200 |

6. Set the **gateway IP address** by replacing the last octet of your virtual switch IP address family with a new numerical value. For example, replace last octet with 1 and get the address 172.20.0.1. Run the following `New-NetIPAddress` command to set the new gateway IP address, replacing the placeholder values:

PowerShell

```

New-NetIPAddress -IPAddress {gatewayIp} -PrefixLength 24 -
InterfaceIndex {interfaceIndex}

```

Running this command should output information similar to the following example:

```
PS C:\WINDOWS\system32> New-NetIPAddress -IPAddress 172.20.0.1 -PrefixLength 24 -InterfaceIndex 61

IPAddress      : 172.20.0.1
InterfaceIndex : 61
InterfaceAlias : vEthernet (EFW)
AddressFamily   : IPv4
Type           : Unicast
PrefixLength   : 24
PrefixOrigin    : Manual
SuffixOrigin    : Manual
AddressState    : Tentative
ValidLifetime   :
PreferredLifetime :
SkipAsSource    : False
PolicyStore     : ActiveStore

IPAddress      : 172.20.0.1
InterfaceIndex : 61
InterfaceAlias : vEthernet (EFW)
AddressFamily   : IPv4
Type           : Unicast
PrefixLength   : 24
PrefixOrigin    : Manual
SuffixOrigin    : Manual
AddressState    : Invalid
ValidLifetime   :
PreferredLifetime :
SkipAsSource    : False
PolicyStore     : PersistentStore
```



7. Create a Network Address Translation (NAT) object that translates an internal network address to an external network. Use the same IPv4 family address from previous steps. Based on the table from step six, the **NAT IP address** corresponds to the original IP address family, except that the last octet is replaced with a new numerical value, for example 0. Run the following [New-NetNat](#) command to set the NAT IP address, replacing the placeholder values:

PowerShell

```
New-NetNat -Name "{switchName}" -InternalIPInterfaceAddressPrefix "{natIp}/24"
```

Running this command should output information similar to the following example:

```
PS C:\WINDOWS\system32> New-NetNat -Name "vEthernet (EFW)" -InternalIPInterfaceAddressPrefix "172.20.0.0/24"

Name          : vEthernet (EFW)
ExternalIPInterfaceAddressPrefix :
InternalIPInterfaceAddressPrefix : 172.20.0.0/24
IcmpQueryTimeout       : 30
TcpEstablishedConnectionTimeout : 1800
TcpTransientConnectionTimeout : 120
TcpFilteringBehavior   : AddressDependentFiltering
UdpFilteringBehavior   : AddressDependentFiltering
UdpIdleSessionTimeout  : 120
UdpInboundRefresh      : False
Store              : Local
Active             : True
```



The switch is now created. Next, you'll set up the DNS.

Create DHCP Server

Note

It is possible to continue the installation without a DHCP server as long as the EFLOW VM is deployed using Static IP parameters (`ip4Address`, `ip4GatewayAddress`, `ip4PrefixLength`). If dynamic IP allocation will be used, ensure to continue with the DHCP server installation.

Warning

Authorization might be required to deploy a DHCP server in a corporate network environment. Check if the virtual switch configuration complies with your corporate network's policies. For more information, see [Deploy DHCP Using Windows PowerShell](#).

1. Check if the DHCP Server feature is installed on the host machine. Look for the **Install State** column. If the value is "Installed", you can skip the following step.

```
PowerShell
```

```
Get-WindowsFeature -Name 'DHCP'
```

2. If the DHCP server isn't already installed, do so by running the following command:

```
PowerShell
```

```
Install-WindowsFeature -Name 'DHCP' -IncludeManagementTools
```

3. Add the DHCP Server to the default local security groups and restart the server.

```
PowerShell
```

```
netsh dhcp add securitygroups  
Restart-Service dhcpserver
```

You'll receive the following warning messages while the DHCP server is starting up:

```
WARNING: Waiting for service 'DHCP Server (dhcpserver)' to start...
```

4. To configure the DHCP server range of IPs to be made available, you'll need to set an IP address as the **start IP** and an IP address as the **end IP**. This range is defined by the **StartRange** and the **EndRange** parameters in the [Add-DhcpServerv4Scope](#) command. You'll also need to set the subnet mask when running this command,

which will be 255.255.255.0. Based on the IP address templates and examples in the table from the previous section, setting the **StartRange** as 169.254.229.100 and the **EndRange** as 169.254.229.200 will make 100 IP addresses available. Run the following command, replacing the placeholders with your own values:

PowerShell

```
Add-DhcpServerV4Scope -Name "AzureIoTEdgeScope" -StartRange {startIp} -  
EndRange {endIp} -SubnetMask 255.255.255.0 -State Active
```

This command should produce no output.

5. Assign the **NAT** and **gateway IP** addresses you created in the earlier section to the DHCP server, and restart the server to load the configuration. The first command should produce no output, but restarting the DHCP server should output the same warning messages that you received when you did so in the third step of this section.

PowerShell

```
Set-DhcpServerV4OptionValue -ScopeID {startIp} -Router {gatewayIp}  
Restart-service dhcpserver
```

Next steps

Follow the steps in [Install and provision Azure IoT Edge for Linux on a Windows device](#) to set up a device with IoT Edge for Linux on Windows.

Azure IoT Edge for Linux on Windows virtual multiple NIC configurations

Article • 05/31/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

By default, the Azure IoT Edge for Linux on Windows (EFLOW) virtual machine has a single network interface card (NIC) assigned. However, you can configure the EFLOW VM with multiple network interfaces by using the EFLOW support for attaching multiple network interfaces to the virtual machine. This functionality may be helpful in numerous scenarios where you may have a networking division or separation into different networks or zones. In order to connect the EFLOW virtual machine to the different networks, you may need to attach different network interface cards to the EFLOW virtual machine.

This article describes how to configure the Azure IoT Edge for Linux on Windows VM to support multiple NICs and connect to multiple networks. This process is divided into the following steps:

- Create and assign a virtual switch
- Create and assign a network endpoint
- Check the VM network configurations

For more information about networking concepts and configurations, see [Azure IoT Edge for Linux on Windows Networking](#) and [How to configure Azure IoT Edge for Linux on Windows networking](#).

Prerequisites

- A Windows device with EFLOW already set up. For more information on EFLOW installation and configuration, see [Create and provision an IoT Edge for Linux on Windows device using symmetric keys](#).
- Virtual switch different from the default one used during EFLOW installation. For more information on creating a virtual switch, see [Create a virtual switch for Azure](#)

Create and assign a virtual switch

During the EFLOW VM deployment, the VM had a switch assigned for all communications between the Windows host OS and the virtual machine. You always use the switch for VM lifecycle management communications, and it's not possible to delete it.

The following steps in this section show how to assign a network interface to the EFLOW virtual machine. Ensure that the virtual switch and the networking configuration align with your networking environment. For more information about networking concepts like type of switches, DHCP and DNS, see [Azure IoT Edge for Linux on Windows networking](#).

1. Open an elevated *PowerShell* session by starting with **Run as Administrator**.
2. Check that the virtual switch you assign to the EFLOW VM is available.

```
PowerShell  
  
Get-VMSwitch -Name "{switchName}" -SwitchType {switchType}
```

3. Assign the virtual switch to the EFLOW VM.

```
PowerShell  
  
Add-EflowNetwork -vSwitchName "{switchName}" -vSwitchType {switchType}
```

For example, if you wanted to assign the external virtual switch named **OnlineExt**, you should use the following command

```
PowerShell  
  
Add-EflowNetwork -vSwitchName "OnlineExt" -vSwitchType "External"
```

```
PS C:\WINDOWS\system32> Add-EflowNetwork -vSwitchName "OnlineExt" -vSwitchType "External"
[07/20/2022 08:22:59] Checking for virtual switch with name 'OnlineExt'
- The virtual switch 'OnlineExt' of type 'External' is present
[07/20/2022 08:23:01] Creating vnet (name: OnlineExt)

Name      AllocationMethod Cidr Type
----      -----          --  --
OnlineExt                         External
```

4. Check that you correctly assigned the virtual switch to the EFLOW VM.

```
PowerShell
```

```
Get-EflowNetwork -vSwitchName "{switchName}"
```

For more information about attaching a virtual switch to the EFLOW VM, see [PowerShell functions for Azure IoT Edge for Linux on Windows](#).

Create and assign a network endpoint

Once you successfully assign the virtual switch to the EFLOW VM, create a networking endpoint assigned to virtual switch to finalize the network interface creation. If you're using Static IP, ensure to use the appropriate parameters: *ip4Address*, *ip4GatewayAddress* and *ip4PrefixLength*.

1. Open an elevated *PowerShell* session by starting with **Run as Administrator**.
2. Create the EFLOW VM network endpoint
 - If you're using DHCP, you don't need Static IP parameters.

```
PowerShell
```

```
Add-EflowVmEndpoint -vSwitchName "{switchName}" -vEndpointName "{EndpointName}"
```

- If you're using Static IP

```
PowerShell
```

```
Add-EflowVmEndpoint -vSwitchName "{switchName}" -vEndpointName "{EndpointName}" -ip4Address "{staticIp4Address}" -
ip4GatewayAddress "{gatewayIp4Address}" -ip4PrefixLength "{prefixLength}"
```

For example, if you wanted to create and assign the **OnlineEndpoint** endpoint with the external virtual switch named **OnlineExt**, and Static IP configurations (*ip4Address=192.168.0.103*, *ip4GatewayAddress=192.168.0.1*, *ip4PrefixLength=24*) you should use the following command:

PowerShell

```
Add-EflowVmEndpoint -vSwitchName "OnlineExt" -vEndpointName "OnlineEndpoint" -ip4Address "192.168.0.103" -ip4GatewayAddress "192.168.0.1" -ip4PrefixLength "24"
```

```
PS C:\WINDOWS\system32> Add-EflowVmEndpoint -vSwitchName "OnlineExt" -vEndpointName "OnlineEndpoint" -ip4Address "192.168.0.103" -ip4GatewayAddress "192.168.0.1" -ip4PrefixLength "24"

Name          : DESKTOP-FCABRER-EFLOW-OnlineExt-OnlineEndpoint
MacAddress    : 00:15:5d:eb:e0:57
HealthStatus   : currentState:OK previousState:OK
IpConfiguration : @{Address=192.168.0.103; PrefixLength=24; Gateway=192.168.0.1; AllocationMethod=Static; SubNet=OnlineExt}
GuestInterfaceName : eth1
```

3. Check that you correctly created the network endpoint and assigned it to the EFLOW VM. You should see two network interfaces assigned to the virtual machine.

PowerShell

```
Get-EflowVmEndpoint
```

```
PS C:\WINDOWS\system32> Get-EflowVmEndpoint

Name          : DESKTOP-FCABRER-EFLOWInterface
MacAddress    : 00:15:5d:2a:07:af
HealthStatus   : currentState:OK previousState:OK
IpConfiguration : @{Address=172.24.42.58; PrefixLength=20; Gateway=; AllocationMethod=Dynamic; SubNet=Default Switch}
GuestInterfaceName : eth0

Name          : DESKTOP-FCABRER-EFLOW-OnlineExt-OnlineEndpoint
MacAddress    : 00:15:5d:eb:e1:29
HealthStatus   : currentState:OK previousState:OK
IpConfiguration : @{Address=192.168.0.103; PrefixLength=24; Gateway=192.168.0.1; AllocationMethod=Static; SubNet=OnlineExt}
GuestInterfaceName : eth1
```

For more information about creating and attaching a network endpoint to the EFLOW VM, see [PowerShell functions for Azure IoT Edge for Linux on Windows](#).

Check the VM network configurations

The final step is to make sure the networking configurations applied correctly and the EFLOW VM has the new network interface configured. The new interface shows up as "eth1" if it's the first extra interface added to the VM.

1. Open PowerShell in an elevated session. You can do so by opening the **Start** pane on Windows and typing in "PowerShell". Right-click the **Windows PowerShell** app that shows up and select **Run as administrator**.

2. Connect to the EFLOW VM.

PowerShell

[Connect-EflowVm](#)

3. Once inside the VM, check the network interfaces and their configurations using the *ifconfig* command.

Bash

`ifconfig`

The default interface **eth0** is the one used for all the VM management. You should see another interface, like **eth1**, which is the new interface you assigned to the VM. Following the examples, if you previously assigned a new endpoint with the static IP 192.168.0.103 you should see the interface **eth1** with the *inet addr*: 192.168.0.103.

```
iotedge-user@DESKTOP-FCABRER-EFLOW [ ~ ]$ ifconfig
docker0  Link encap:Ethernet HWaddr 02:42:5f:44:d8:e4
          inet addr:172.17.0.1 Bcast:172.17.255.255 Mask:255.255.0.0
                  UP BROADCAST MULTICAST MTU:1500 Metric:1
                  RX packets:0 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:0
                  RX bytes:0 (0.0 B) TX bytes:0 (0.0 B)

eth0      Link encap:Ethernet HWaddr 00:15:5d:2a:07:af
          inet addr:172.20.125.52 Bcast:172.20.127.255 Mask:255.255.240.0
          inet6 addr: fe80::215:5dff:fe2a:7af/64 Scope:Link
                  UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
                  RX packets:23108 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:479 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:1000
                  RX bytes:2172239 (2.1 MB) TX bytes:81864 (81.8 KB)

eth1      Link encap:Ethernet HWaddr 00:15:5d:eb:e0:57
          inet addr:192.168.0.103 Bcast:192.168.0.255 Mask:255.255.255.0
          inet6 addr: fe80::215:5dff:feebe057/64 Scope:Link
                  UP BROADCAST RUNNING MULTICAST MTU:1500 Metric:1
                  RX packets:4206 errors:0 dropped:0 overruns:0 frame:0
                  TX packets:20 errors:0 dropped:0 overruns:0 carrier:0
                  collisions:0 txqueuelen:1000
                  RX bytes:400021 (400.0 KB) TX bytes:1328 (1.3 KB)
```

Next steps

Follow the steps in [How to configure networking for Azure IoT Edge for Linux on Windows](#) to make sure you applied all the networking configurations correctly.

How to configure Azure IoT Edge for Linux on Windows on a DMZ

Article • 05/31/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This article describes how to configure the Azure IoT Edge for Linux (EFLOW) virtual machine (VM) to support multiple network interface cards (NICs) and connect to multiple networks. By enabling multiple NIC support, applications running on the EFLOW VM can communicate with devices connected to the offline network, while using IoT Edge to send data to the cloud.

Prerequisites

- A Windows device with EFLOW already set up. For more information on EFLOW installation and configuration, see [Create and provision an IoT Edge for Linux on Windows device using symmetric keys](#).
- A virtual switch different from the default one used during EFLOW installation. For more information on creating a virtual switch, see [Create a virtual switch for Azure IoT Edge for Linux on Windows](#).

Industrial scenario

Industrial IoT is overtaking the era of information technology (IT) and operational technology (OT) convergence. However, making traditional OT assets more intelligent with IT technologies also means a larger exposure to cyber attacks. This scenario is one of the main reasons why multiple environments are designed using demilitarized zones, also known as, DMZs.

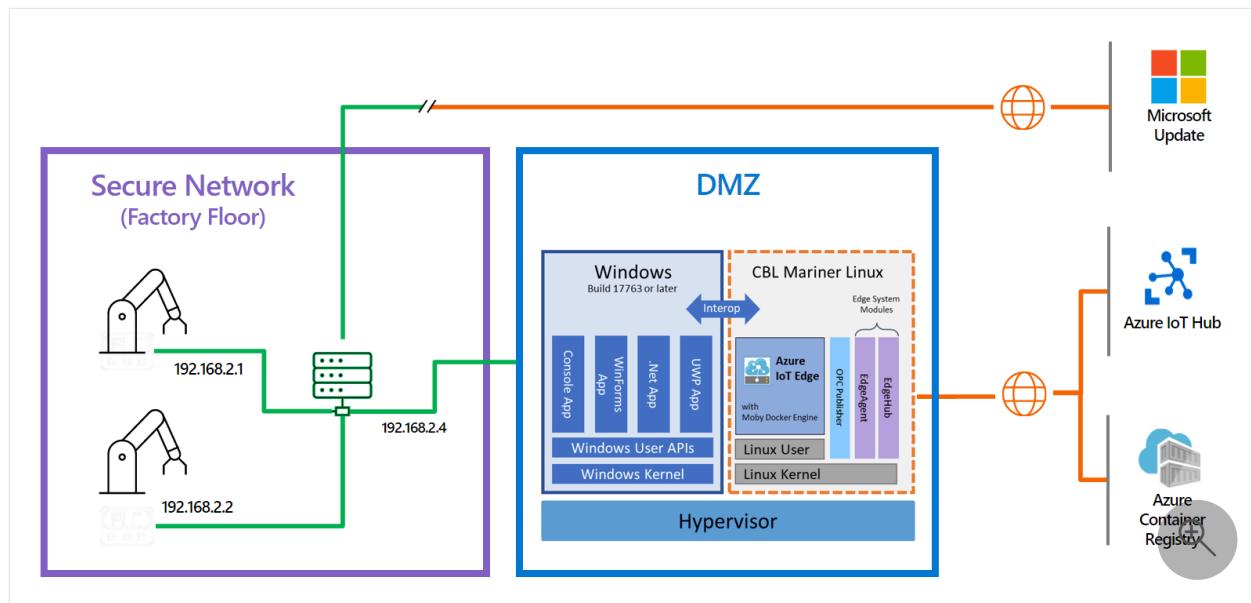
Imagine a workflow scenario where you have a networking configuration divided into two different networks or zones. In the first zone, you may have a secure network defined as the offline network. The offline network has no internet connectivity and is limited to internal access. In the second zone, you may have a demilitarized zone (DMZ),

in which you may have a couple of devices that have limited internet connectivity. When moving the workflow to run on the EFLOW VM, you may have problems accessing the different networks since the EFLOW VM by default has only one NIC attached.

In this scenario, you have an environment with some devices like programmable logic controllers (PLCs) or open platform communications unified architecture (OPC UA)-compatible devices connected to the offline network, and you want to upload all the devices' information to Azure using the OPC Publisher module running on the EFLOW VM.

Since the EFLOW host device and the PLC or OPC UA devices are physically connected to the offline network, you can use the [Azure IoT Edge for Linux on Windows virtual multiple NIC configurations](#) to connect the EFLOW VM to the offline network. By using an *external virtual switch*, you can connect the EFLOW VM to the offline network and directly communicate with other offline devices.

For the other network, the EFLOW host device is physically connected to the DMZ (online network) with internet and Azure connectivity. Using an *internal or external switch*, you can connect the EFLOW VM to Azure IoT Hub using IoT Edge modules and upload the information sent by the offline devices through the offline NIC.



Scenario summary

Secure network:

- No internet connectivity - access restricted.
- PLCs or OPC UA-compatible devices connected.
- EFLOW VM connected using an External virtual switch.

DMZ:

- Internet connectivity - Azure connection allowed.
- EFLOW VM connected to Azure IoT Hub, using either an Internal/External virtual switch.
- OPC Publisher running as a module inside the EFLOW VM used to publish data to Azure.

Configure VM network virtual switches

The following steps are specific for the networking described in the example scenario. Ensure that the virtual switches used and the configurations used align with your networking environment.

Note

The steps in this article assume that the EFLOW VM was deployed with an *external virtual switch* connected to the *secure network (offline)*. You can change the following steps to the specific network configuration you want to achieve. For more information about EFLOW multiple NICs support, see [Azure IoT Edge for Linux on Windows virtual multiple NIC configurations](#).

To finish the provisioning of the EFLOW VM and communicate with Azure, you need to assign another NIC that is connected to the DMZ network (online).

For this scenario, you assign an *external virtual switch* connected to the DMZ network. For more information, review [Create a virtual switch for Hyper-V virtual machines](#).

To create an external virtual switch, follow these steps:

1. Open Hyper-V Manager.
2. In **Actions**, select **Virtual Switch Manager**.
3. In **Virtual Switches**, select **New Virtual network switch**.
4. Choose type **External** then select **Create Virtual Switch**.
5. Enter a name that represents the secure network. For example, *OnlineOPCUA*.
6. Under **Connection Type**, select **External Network** then choose the *network adapter* connected to your DMZ network.
7. Select **Apply**.

Once the external virtual switch is created, you need to attach it to the EFLOW VM using the following steps. If you need to attach multiple NICs, see [EFLOW Multiple NICs](#).

For the custom new *external virtual switch* you created, use the following PowerShell commands to:

1. Attach the switch to your EFLOW VM.

```
PowerShell
```

```
Add-EflowNetwork -vswitchName "OnlineOPCUA" -vswitchType "External"
```

```
PS C:\WINDOWS\system32> Add-EflowNetwork -vswitchName "OnlineOPCUA" -vswitchType "External"
[10/21/2021 13:10:05] Creating vnet (name: OnlineOPCUA)

Name      AllocationMethod Cidr Type
----      -----          --  --
OnlineOPCUA          External
```

2. Set a static IP.

```
PowerShell
```

```
Add-EflowVmEndpoint -vswitchName "OnlineOPCUA" -vEndpointName
"OnlineEndpoint" -ip4Address 192.168.0.103 -ip4PrefixLength 24 -
ip4GatewayAddress 192.168.0.1
```

```
PS C:\WINDOWS\system32> Add-EflowVmEndpoint -vSwitchName "OnlineOPCUA" -vEndpointName "OnlineEndpoint" -ip4Address "192.168.0.103" -ip4PrefixLength 24 -ip4GatewayAddress "192.168.0.1"
Name           MacAddress     HealthStatus   IpConfiguration
--           -----          -----        -----
DESKTOP-SFRE9NQ-EFLOW-OnlineOPCUA-OnlineEndpoint 00:15:5d:54:34:46 currentState:OK previousState:OK @{Address=; PrefixLength=; AllocationMethod=Dynamic; SubNet=0...}
```

Once complete, you have the *OnlineOPCUA* switch assigned to the EFLOW VM. To check the multiple NIC attachment, use the following steps:

1. Open an elevated PowerShell session by starting with **Run as Administrator**.
2. Connect to the EFLOW virtual machine.

```
PowerShell
```

```
Connect-EflowVm
```

3. Once you're in your VM, list all the network interfaces assigned to the EFLOW virtual machine.

```
Bash
```

```
ifconfig
```

4. Review the IP configuration and verify you see the *eth0* interface (connected to the secure network) and the *eth1* interface (connected to the DMZ network).

```
eth0      Link encap:Ethernet HWaddr 00:15:5d:dd:b7:a9
          inet addr:192.168.2.4  Bcast:192.168.2.255  Mask:255.255.255.0
                      inet6 addr: fe80::215:5dff:fedd:b7a9/64 Scope:Link
                        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
                        RX packets:42 errors:0 dropped:0 overruns:0 frame:0
                        TX packets:43 errors:0 dropped:0 overruns:0 carrier:0
                        collisions:0 txqueuelen:1000
                        RX bytes:6247 (6.2 KB)  TX bytes:6408 (6.4 KB)

eth1      Link encap:Ethernet HWaddr 00:15:5d:54:34:46
          inet addr:192.168.0.103  Bcast:192.168.0.255  Mask:255.255.255.0
                      inet6 addr: fe80::215:5dff:fe54:3446/64 Scope:Link
                        UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
                        RX packets:167 errors:0 dropped:0 overruns:0 frame:0
                        TX packets:41 errors:0 dropped:0 overruns:0 carrier:0
                        collisions:0 txqueuelen:1000
                        RX bytes:41236 (41.2 KB)  TX bytes:10291 (10.2 KB)
```

Configure VM network routing

When using the EFLOW multiple NICs feature, you may want to set up the different route priorities. By default, EFLOW creates one *default* route per *ethX* interface assigned to the VM. EFLOW assigns the default route a random priority. If all interfaces are connected to the internet, random priorities may not be a problem. However, if one of the NICs is connected to an *offline* network, you may want to prioritize the *online* NIC over the *offline* NIC to get the EFLOW VM connected to the internet.

EFLOW uses the [route ↗](#) service to manage the network routing alternatives. In order to check the available EFLOW VM routes, use the following steps:

1. Open an elevated PowerShell session by starting with **Run as Administrator**.
2. Connect to the EFLOW virtual machine.

```
PowerShell
```

```
Connect-EflowVm
```

3. Once you're in your VM, list all the network routes configured in the EFLOW virtual machine.

```
Bash
```

```
sudo route
```

```
iotedge-user@DESKTOP-SFRE9NQ-EFLOW [ ~ ]$ sudo route
Kernel IP routing table
Destination     Gateway         Genmask        Flags Metric Ref  Use Iface
default         192.168.2.2   0.0.0.0       UG    0      0      0 eth1
default         192.168.0.1   0.0.0.0       UG    1024   0      0 eth0
172.17.0.0     *              255.255.0.0   U     0      0      0 docker0
192.168.0.0    *              255.255.255.0  U     0      0      0 eth0
192.168.0.1    *              255.255.255.255 UH    1024   0      0 eth0
192.168.2.0    *              255.255.255.0  U     0      0      0 eth1
```

💡 Tip

The previous image shows the route command output with the two NIC's assigned (*eth0* and *eth1*). The virtual machine creates two different *default* destinations rules with different metrics. A lower metric value has a higher priority. This routing table will vary depending on the networking scenario configured in the previous steps.

Static routes configuration

Every time EFLOW VM starts, the networking services recreates all routes, and any previously assigned priority could change. To work around this issue, you can assign the desired priority for each route every time the EFLOW VM starts. You can create a service that executes on every VM boot and uses the `route` command to set the desired route priorities.

First, create a bash script that executes the necessary commands to set the routes. For example, following the networking scenario mentioned earlier, the EFLOW VM has two NICs (offline and online networks). NIC *eth0* is connected using the gateway IP `xxx.xxx.xxx.xxx`. NIC *eth1* is connected using the gateway IP `yyy.yyy.yyy.yyy`.

The following script resets the *default* routes for both *eth0* and **eth1* then adds the routes with the desired `<number>` metric. Remember that *a lower metric value has higher priority*.

Bash

```
#!/bin/sh

# Wait 30s for the interfaces to be up
sleep 30

# Delete previous eth0 route and create a new one with desired metric
sudo ip route del default via xxx.xxx.xxx.xxx dev eth0
sudo route add -net default gw xxx.xxx.xxx.xxx netmask 0.0.0.0 dev eth0
metric <number>
```

```
# Delete previous eth1 route and create a new one with desired metric
sudo ip route del default via yyy.yyy.yyy.yyy dev eth1
sudo route add -net default gw yyy.yyy.yyy.yyy netmask 0.0.0.0 dev eth1
metric <number>
```

You can use the previous script to create your own custom script specific to your networking scenario. Once script is defined, save it, and assign execute permission. For example, if the script name is *route-setup.sh*, you can assign execute permission using the command `sudo chmod +x route-setup.sh`. You can test if the script works correctly by executing it manually using the command `sudo sh ./route-setup.sh` and then checking the routing table using the `sudo route` command.

The final step is to create a Linux service that runs on startup, and executes the bash script to set the routes. You have to create a *systemd* unit file to load the service. The following is an example of that file.

```
systemd

[Unit]
after=network

[Service]
Type=simple
ExecStart=/bin/bash /home/iotedge-user/route-setup.sh

[Install]
WantedBy=default.target
```

To check the service works, reboot the EFLOW VM (`Stop-EflowVm` & `Start-EflowVm`) then `Connect-EflowVm` to connect to the VM. List the routes using `sudo route` and verify they're correct. You should be able to see the new *default* rules with the assigned metric.

Next steps

Follow the steps in [How to configure networking for Azure IoT Edge for Linux on Windows](#) to verify your networking configurations were applied correctly.

Share a Windows folder with Azure IoT Edge for Linux on Windows

Article • 06/04/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

The Azure IoT Edge for Linux on Windows (EFLOW) virtual machine is isolated from the Windows host OS and the virtual machine doesn't have access to the host file system. By default, the EFLOW virtual machine has its own file system and has no access to the folders or files on the host computer. The *EFLOW file and folder sharing mechanism* provides a way to share Windows files and folders to the CBL-Mariner Linux EFLOW VM.

This article shows you how to enable the folder sharing between the Windows host OS and the EFLOW virtual machine.

Prerequisites

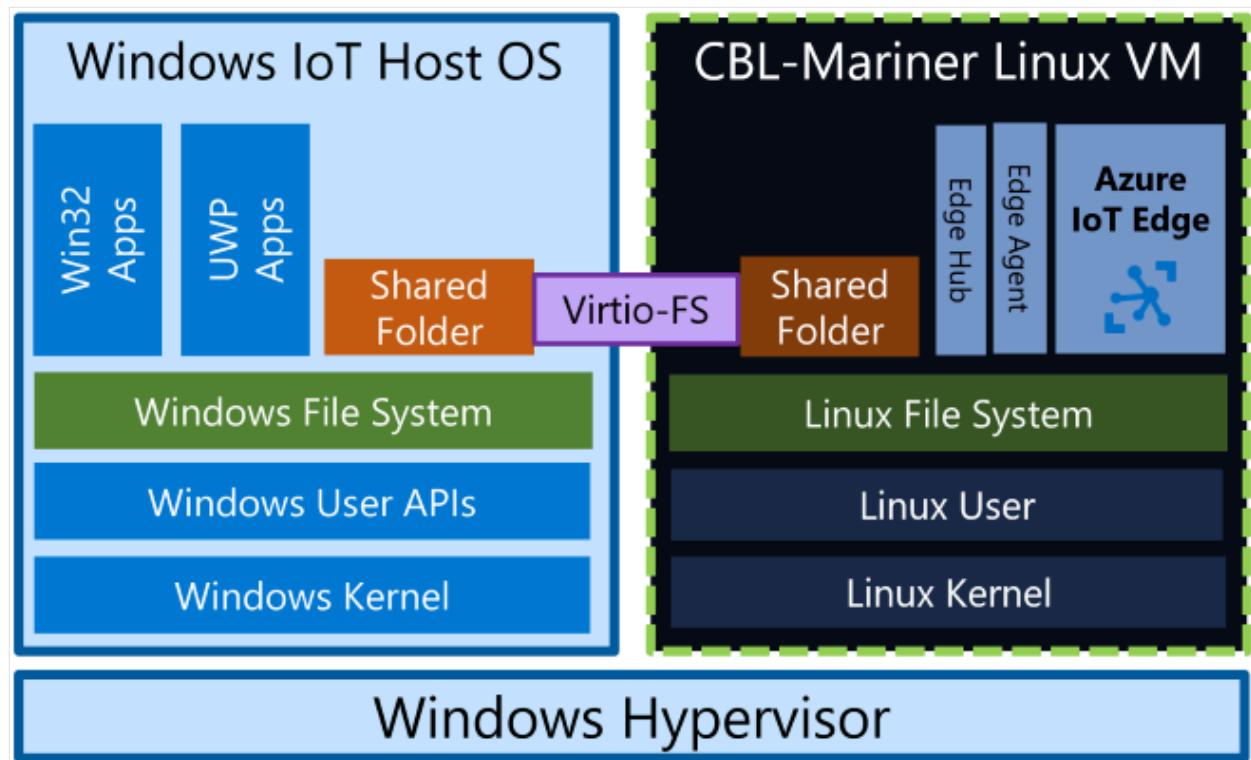
- Azure IoT Edge for Linux on Windows 1.4.4 LTS update or higher. For more information about EFLOW release notes, see [EFLOW Releases](#).
- A machine with an x64/x86 processor.
- Windows 10/11 (21H2) or higher with [November 2022](#) update applied.

If you don't have an EFLOW device ready, you should create one before continuing with this guide. Follow the steps in [Create and provision an IoT Edge for Linux on Windows device using symmetric keys](#) to install, deploy, and provision EFLOW.

How it works?

The Azure IoT Edge for Linux on Windows file and folder sharing mechanism is implemented using [virtiofs](#) technology. *Virtiofs* is a shared file system that lets virtual machines access a directory tree on the host OS. Unlike other approaches, it's designed to offer local file system semantics and performance. *Virtiofs* isn't a network file system repurposed for virtualization. It's designed to take advantage of the locality of virtual

machines and the hypervisor. It takes advantage of the virtual machine's co-location with the hypervisor to avoid overhead associated with network file systems.



Only Windows folders can be shared to the EFLOW Linux VM and not the other way. Also, for security reasons, when setting the folder sharing mechanism, the user must provide a *root folder* and all the shared folders must be under that *root folder*.

Before starting with the adding and removing share mechanisms, let's define four concepts:

- **Root folder:** Windows folder that is the root path containing subfolders to be shared to the EFLOW VM. The root folder isn't shared to the EFLOW VM. Only the subfolders under the root folder are shared to the EFLOW VM.
- **Shared folder:** A Windows folder that's under the *root folder* and is shared with the EFLOW VM. All the content inside this folder is shared with the EFLOW VM.
- **Mounting point:** Path inside the EFLOW VM where the Windows folder content is placed.
- **Mounting option:** *Read-only* or *read and write* access. Controls the file access of the mounted folder inside the EFLOW VM.

Add shared folders

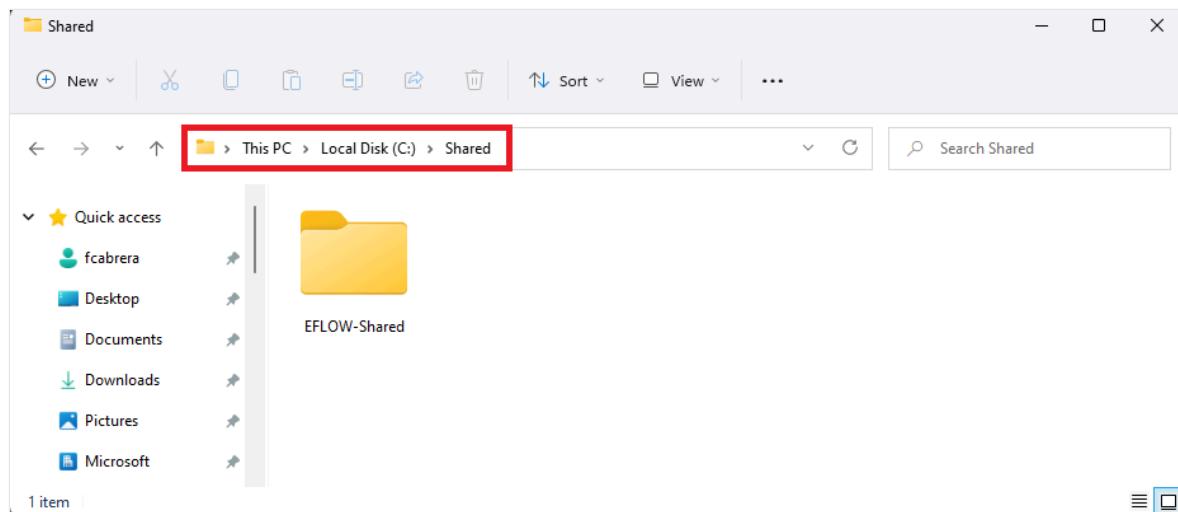
The following steps provide example EFLOW PowerShell commands to share one or more Windows host OS folders with the EFLOW virtual machine.

⚠ Note

If you're using Windows 10, ensure to reboot your Windows host OS after your fresh MSI installation or update before adding the Windows shared folders to the EFLOW VM.

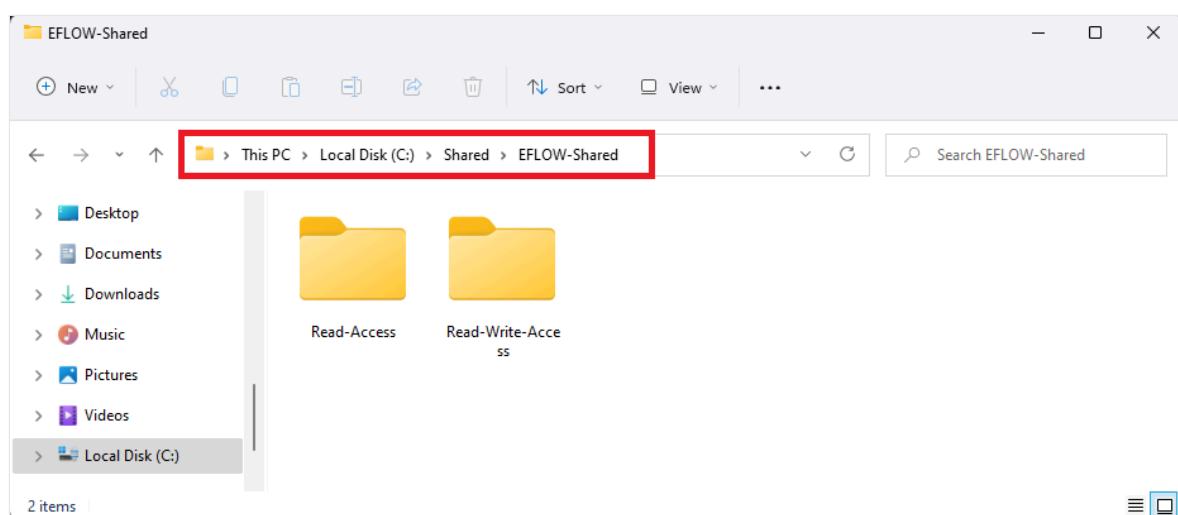
1. Start by creating a new root shared folder. Go to **File Explorer** and choose a location for the *root folder* and create the folder.

For example, create a *root folder* under C:\Shared named **EFLOW-Shared**.



2. Create one or more *shared folders* to be shared with the EFLOW virtual machine. Shared folders should be created under the *root folder* from the previous step.

For example, create two folders one named **Read-Access** and one named **Read-Write-Access**.



3. Within the *Read-Access* shared folder, create a sample file that we'll later read inside the EFLOW virtual machine.

For example, using a text editor, create a file named *Hello-World.txt* within the *Read-Access* folder and save some text in the file.

4. Using a text editor, create the shared folder configuration file. This file contains the information about the folders to be shared with the EFLOW VM including the mounting points and the mounting options. For more information about the JSON configuration file, see [PowerShell functions for IoT Edge for Linux on Windows](#).

For example, using the previous scenario, we'll share the two *shared folders* we created under the *root folder*.

- *Read-Access* shared folder is mounted in the EFLOW virtual machine under the path */tmp/host-read-access* with *read-only* access.
- *Read-Write-Access* shared folder is mounted in the EFLOW virtual machine under the path */tmp/host-read-write-access* with *read and write* access.

Create the JSON configuration file named **sharedFolders.json** within the *root folder EFLOW-Shared* with the following contents:

```
JSON

[
  {
    "sharedFolderRoot": "C:\\Shared\\EFLOW-Shared",
    "sharedFolders": [
      {
        "hostFolderPath": "Read-Access",
        "readOnly": true,
        "targetFolderOnGuest": "/tmp/host-read-access"
      },
      {
        "hostFolderPath": "Read-Write-Access",
        "readOnly": false,
        "targetFolderOnGuest": "/tmp/host-read-write-access"
      }
    ]
  }
]
```

5. Open an elevated *PowerShell* session by starting with **Run as Administrator**.
6. Create the shared folder assignation using the configuration file (*sharedFolders.json*) previously created.

```
PowerShell

Add-EflowVmSharedFolder -sharedFoldersJsonPath "C:\\Shared\\EFLOW-
```

```
Shared\sharedFolders.json"
```

7. Once the cmdlet finished, the EFLOW virtual machine should have access to the shared folders. Connect to the EFLOW virtual machine and check the folders are correctly shared.

```
PowerShell
```

```
Connect-EflowVm
```

8. Go to the *Read-Access* shared folder (mounted under */tmp/host-read-access*) and check the content of the *Hello-World.txt* file.

 **Note**

By default, all shared folders are shared under *root* ownership. To access the folder, you should log in as root using `sudo su` or change the folder ownership to *iotedge-user* using `chown` command.

```
Bash
```

```
sudo su
cd /tmp/host-read-access
cat Hello-World.txt
```

If everything was successful, you should be able to see the contents of the *Hello-World.txt* file within the EFLOW virtual machine. Verify write access by creating a file inside the */tmp/host-read-write-access* and then checking the contents of the new created file inside the *Read-Write-Access* Windows host folder.

Check shared folders

The following steps provide example EFLOW PowerShell commands to check the Windows shared folders and options (access permissions and mounting point) with the EFLOW virtual machine.

1. Open an elevated PowerShell session by starting with **Run as Administrator**.
2. List the information of the Windows shared folders under the *root folder*. For example, using the scenario in the previous section, we can list the information of both *Read-Access* and *Read-Write-Access* shared folders.

PowerShell

```
Get-EflowVmSharedFolder -sharedfolderRoot "C:\Shared\EFLOW-Shared" -  
hostFolderPath @("Read-Access", "Read-Write-Access")
```

For more information about the `Get-EflowVmSharedFolder` cmdlet, see [PowerShell functions for IoT Edge for Linux on Windows](#).

Remove shared folders

The following steps provide example EFLOW PowerShell commands to stop sharing a Windows shared folder with the EFLOW virtual machine.

1. Open an elevated PowerShell session by starting with **Run as Administrator**.
2. Stop sharing the folder named *Read-Access* under the **Root folder** with the EFLOW virtual machine.

PowerShell

```
Remove-EflowVmSharedFolder -sharedfolderRoot "C:\Shared\EFLOW-Shared" -  
hostFolderPath "Read-Access"
```

For more information about the `Remove-EflowVmSharedFolder` cmdlet, see [PowerShell functions for IoT Edge for Linux on Windows](#).

Next steps

Follow the steps in [Common issues and resolutions for Azure IoT Edge for Linux on Windows](#) to troubleshoot any issues encountered when setting up IoT Edge for Linux on Windows.

Tutorial: Develop IoT Edge modules with Linux containers using IoT Edge for Linux on Windows

Article • 03/27/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

This tutorial walks through developing, debugging and deploying your own code to an IoT Edge device using IoT Edge for Linux on Windows and Visual Studio 2022. You'll learn the most common developer scenario for IoT Edge solutions by deploying a **C# module to a Linux device**. You'll deploy and debug a custom Azure IoT Edge module running in a Linux container on Windows (EFLOW). Even if you plan on using a different language or deploying an Azure service, this tutorial is still useful to learn about the development tools and concepts.

This article includes steps for two IoT Edge development tools:

- Command line interface (CLI) is the preferred tool for development.
- **Azure IoT Edge tools for Visual Studio** extension. The extension is in [maintenance mode](#).

Use the tool selector button at the beginning of this article to select the tool version.

In this tutorial, you learn how to:

- ✓ Set up your development machine.
- ✓ Use the IoT Edge tools for Visual Studio Code to create a new project.
- ✓ Build your project as a container and store it in an Azure container registry.
- ✓ Deploy your code to an IoT Edge device.

Prerequisites

This article assumes that you use a machine running Windows as your development machine. On Windows computers, you can develop either Windows or Linux modules.

This tutorial guides you through the development of **Linux containers**, using [IoT Edge for Linux on Windows](#) for building and deploying the modules.

- Install [IoT Edge for Linux on Windows \(EFLOW\)](#)
- Quickstart: [Deploy your first IoT Edge module to a Windows device](#)
- [.NET Core SDK](#).
- Install or modify Visual Studio 2022 on your development machine. Choose the **Azure development** and **Desktop development with C++** workloads options.

After your Visual Studio 2022 is ready, you also need the following tools and components:

- Download and install [Azure IoT Edge Tools](#) from the Visual Studio Marketplace. You can use the Azure IoT Edge Tools extension to create and build your IoT Edge solution. The preferred development tool is the command-line (CLI) *Azure IoT Edge Dev Tool*. The extension includes the Azure IoT Edge project templates used to create the Visual Studio project. Currently, you need the extension installed regardless of the development tool you use.

 **Important**

The *Azure IoT Edge Tools for VS 2022* extension is in [maintenance mode](#).

The preferred development tool is the command-line (CLI) *Azure IoT Edge Dev Tool*.

 **Tip**

If you are using Visual Studio 2019, download and install [Azure IoT Edge Tools for VS 2019](#) from the Visual Studio marketplace.

- Install the [Azure CLI](#).

Cloud resources:

- A free or standard-tier [IoT hub](#) in Azure.

If you don't have an [Azure subscription](#), create an [Azure free account](#) before you begin.

Key concepts

This tutorial walks through the development of an IoT Edge module. An *IoT Edge module*, or sometimes just *module* for short, is a container with executable code. You can deploy one or more modules to an IoT Edge device. Modules perform specific tasks like ingesting data from sensors, cleaning and analyzing data, or sending messages to an IoT hub. For more information, see [Understand Azure IoT Edge modules](#).

When developing IoT Edge modules, it's important to understand the difference between the development machine and the target IoT Edge device where the module will eventually be deployed. The container that you build to hold your module code must match the operating system (OS) of the *target device*. For example, the most common scenario is someone developing a module on a Windows computer intending to target a Linux device running IoT Edge. In that case, the container operating system would be Linux. As you go through this tutorial, keep in mind the difference between the *development machine OS* and the *container OS*. For this tutorial, you'll be using your Windows host for development and the IoT Edge for Linux on Windows (EFLW) VM for building and deploying the modules.

This tutorial targets devices running IoT Edge with Linux containers. You can use your preferred operating system as long as your development machine runs Linux containers. We recommend using Visual Studio to develop with Linux containers, so that's what this tutorial uses. You can use Visual Studio Code as well, although there are differences in support between the two tools. For more information, see [Develop Azure IoT Edge modules using Visual Studio Code](#).

Set up docker-cli and Docker engine remote connection

IoT Edge modules are packaged as containers, so you need a container engine on your development machine to build and manage them. The EFLW virtual machine already contains an instance of Docker engine, so this tutorial shows you how to remotely connect from the Windows developer machine to the EFLW VM Docker instance. By using this, we remove the dependency on Docker Desktop for Windows.

The first step is to configure docker-cli on the Windows development machine to be able to connect to the remote docker engine.

1. Download the precompiled `docker.exe` version of the docker-cli from [docker-cli Chocolatey](#). You can also download the official `cli` project from [docker/cli GitHub](#) and compile it following the repo instructions.

2. Extract the **docker.exe** to a directory in your development machine. For example,
`C:\Docker\bin`
3. Open **About your PC** -> **System Info** -> **Advanced system settings**
4. Select **Advanced** -> **Environment variables** -> Under **User variables** check **Path**
5. Edit the **Path** variable and add the location of the **docker.exe**
6. Open an elevated PowerShell session
7. Check that Docker CLI is accessible using the command

```
PowerShell  
docker --version
```

If everything was successfully configurated, the previous command should output the docker version, something like *Docker version 20.10.12, build e91ed57*.

The second step is to configure the EFLOW virtual machine Docker engine to accept external connections, and add the appropriate firewall rules.

⚠ Warning

Exposing Docker engine to external connections may increase security risks. This configuration should only be used for development purposes. Make sure to revert the configuration to default settings after development is finished.

1. Open an elevated PowerShell session and run the following commands

```
PowerShell  
  
# Configure the EFLOW virtual machine Docker engine to accept external  
connections, and add the appropriate firewall rules.  
Invoke-EflowVmCommand "sudo iptables -A INPUT -p tcp --dport 2375 -j  
ACCEPT"  
  
# Create a copy of the EFLOW VM _docker.service_ in the system folder.  
Invoke-EflowVmCommand "sudo cp /lib/systemd/system/docker.service  
/etc/systemd/system/docker.service"  
  
# Replace the service execution line to listen for external  
connections.  
Invoke-EflowVmCommand "sudo sed -i 's/-H fd:\:\/\/ -H fd:\\/\/ -H  
tcp:\\/\0.0.0.0:2375/g' /etc/systemd/system/docker.service"  
  
# Reload the EFLOW VM services configurations.  
Invoke-EflowVmCommand "sudo systemctl daemon-reload"  
  
# Reload the Docker engine service.
```

```
Invoke-EflowVmCommand "sudo systemctl restart docker.service"

# Check that the Docker engine is listening to external connections.
Invoke-EflowVmCommand "sudo netstat -lntp | grep dockerd"
```

The following is example output.

Output

```
PS C:\> # Configure the EFLOW virtual machine Docker engine to accept
external connections, and add the appropriate firewall rules.
PS C:\> Invoke-EflowVmCommand "sudo iptables -A INPUT -p tcp --dport
2375 -j ACCEPT"
PS C:\>
PS C:\> # Create a copy of the EFLOW VM docker.service in the system
folder.
PS C:\> Invoke-EflowVmCommand "sudo cp
/lib/systemd/system/docker.service /etc/systemd/system/docker.service"
PS C:\>
PS C:\> # Replace the service execution line to listen for external
connections.
PS C:\> Invoke-EflowVmCommand "sudo sed -i 's/-H fd:\:\/\/ -H fd:\\/\/-H
tcp:\\/\0.0.0:2375/g' /etc/systemd/system/docker.service"
PS C:\>
PS C:\> # Reload the EFLOW VM services configurations.
PS C:\> Invoke-EflowVmCommand "sudo systemctl daemon-reload"
PS C:\>
PS C:\> # Reload the Docker engine service.
PS C:\> Invoke-EflowVmCommand "sudo systemctl restart docker.service"
PS C:\>
PS C:\> # Check that the Docker engine is listening to external
connections.
PS C:\> Invoke-EflowVmCommand "sudo netstat -lntp | grep dockerd"
tcp6      0      0 :::2375          :::*
LISTEN    2790/dockerd
```

2. The final step is to test the Docker connection to the EFLOW VM Docker engine.

First, you need the EFLOW VM IP address.

PowerShell

```
Get-EflowVmAddr
```

 Tip

If the EFLOW VM was deployed without Static IP, the IP address may change across Windows host OS reboots or networking changes. Make sure you are

using the correct EFLOW VM IP address every time you want to establish a remote Docker engine connection.

The following is example output.

Output

```
PS C:\> Get-EflowVmAddr
[03/15/2022 15:22:30] Querying IP and MAC addresses from virtual
machine (DESKTOP-J1842A1-EFLOW)
- Virtual machine MAC: 00:15:5d:6f:da:78
- Virtual machine IP : 172.31.24.105 retrieved directly from virtual
machine
00:15:5d:6f:da:78
172.31.24.105
```

3. Using the obtained IP address, connect to the EFLOW VM Docker engine, and run the Hello-World sample container. Replace <EFLOW-VM-IP> with the EFLOW VM IP address obtained in the previous step.

PowerShell

```
docker -H tcp://<EFLOW-VM-IP>:2375 run --rm hello-world
```

You should see that the container is being downloaded, and after will run and output the following.

Output

```
PS C:\> docker -H tcp://172.31.24.105:2375 run --rm hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
2db29710123e: Pull complete
Digest:
sha256:4c5f3db4f8a54eb1e017c385f683a2de6e06f75be442dc32698c9bbe6c861edd
Status: Downloaded newer image for hello-world:latest
```

Hello from Docker!

This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the

executable that produces the output you are currently reading.

4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:

```
$ docker run -it ubuntu bash
```

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

For more examples and ideas, visit:
<https://docs.docker.com/get-started/>

Create an Azure IoT Edge project

The IoT Edge project template in Visual Studio creates a solution that can be deployed to IoT Edge devices. First you create an Azure IoT Edge solution, and then you generate the first module in that solution. Each IoT Edge solution can contain more than one module.

Important

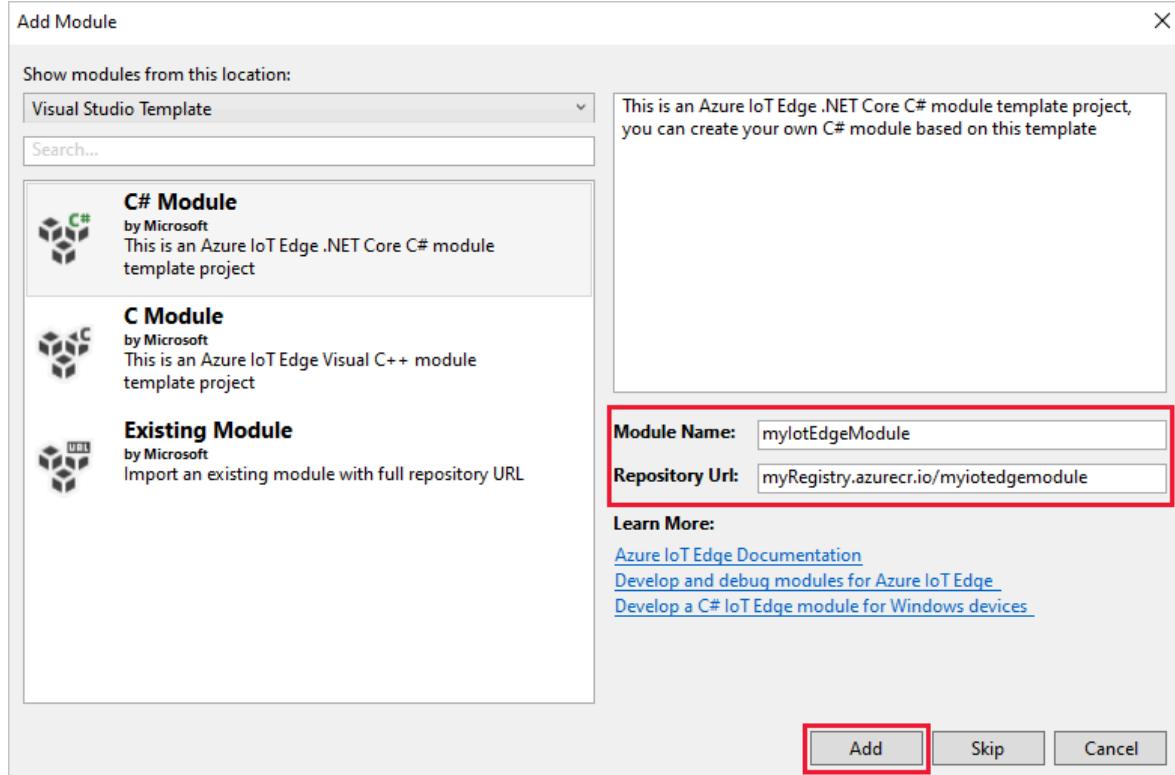
The IoT Edge project structure created by Visual Studio isn't the same as the one in Visual Studio Code.

Currently, the Azure IoT Edge Dev Tool CLI doesn't support creating the Visual Studio project type. You need to use the Visual Studio IoT Edge extension to create the Visual Studio project.

1. In Visual Studio, create a new project.
2. On the **Create a new project** page, search for **Azure IoT Edge**. Select the project that matches the platform (Linux IoT Edge module) and architecture for your IoT Edge device, and select **Next**.
3. On the **Configure your new project** page, enter a name for your project and specify the location, then select **Create**.
4. On the **Add Module** window, select the type of module you want to develop. You can also select **Existing module** to add an existing IoT Edge module to your deployment. Specify your module name and module image repository.
5. In **Repository Url**, provide the name of the module's image repository. Visual Studio autopopulates the module name with **localhost:5000/<your module**

`name`. Replace it with your own registry information. Use `localhost` if you use a local Docker registry for testing. If you use Azure Container Registry, then use the login server from your registry's settings. The login server looks like `<registry name>.azurecr.io`. Only replace the `localhost:5000` part of the string so that the final result looks like `<registry name>.azurecr.io/<your module name>`.

6. Select **Add** to add your module to the project.



ⓘ Note

If you have an existing IoT Edge project, you can change the repository URL by opening the `module.json` file. The repository URL is located in the `repository` property of the JSON file.

Now, you have an IoT Edge project and an IoT Edge module in your Visual Studio solution.

Project structure

In your solution, there are two project level folders including a main project folder and a single module folder. For example, you may have a main project folder named `AzureIoTEdgeApp1` and a module folder named `IoTEdgeModule1`. The main project folder contains your deployment manifest.

The module folder contains a file for your module code, named either `Program.cs` or `main.c` depending on the language you chose. This folder also contains a file named `module.json` that describes the metadata of your module. Various Docker files provide the information needed to build your module as a Windows or Linux container.

Deployment manifest of your project

The deployment manifest you edit is named `deployment.debug.template.json`. This file is a template of an IoT Edge deployment manifest that defines all the modules that run on a device along with how they communicate with each other. For more information about deployment manifests, see [Learn how to deploy modules and establish routes](#).

If you open this deployment template, you see that the two runtime modules, `edgeAgent` and `edgeHub` are included, along with the custom module that you created in this Visual Studio project. A fourth module named `SimulatedTemperatureSensor` is also included. This default module generates simulated data that you can use to test your modules, or delete if it's not necessary. To see how the simulated temperature sensor works, view the [SimulatedTemperatureSensor.csproj source code](#).

Set IoT Edge runtime version

Currently, the latest stable runtime version is 1.4. You should update the IoT Edge runtime version to the latest stable release or the version you want to target for your devices.

1. Open `deployment.debug.template.json` deployment manifest file. The [deployment manifest](#) is a JSON document that describes the modules to be configured on the targeted IoT Edge device.
2. Change the runtime version for the system runtime module images `edgeAgent` and `edgeHub`. For example, if you want to use the IoT Edge runtime version 1.4, change the following lines in the deployment manifest file:

JSON

```
"systemModules": {
    "edgeAgent": {
        //...
        "image": "mcr.microsoft.com/azureiotedge-agent:1.4",
        //...
    "edgeHub": {
        //...
```

```
"image": "mcr.microsoft.com/azureiotedge-hub:1.4",  
//...
```

Develop your module

When you add a new module, it comes with default code that is ready to be built and deployed to a device so that you can start testing without touching any code. The module code is located within the module folder in a file named `Program.cs` (for C#) or `main.c` (for C).

The default solution is built so that the simulated data from the **SimulatedTemperatureSensor** module is routed to your module, which takes the input and then sends it to IoT Hub.

When you're ready to customize the module template with your own code, use the [Azure IoT Hub SDKs](#) to build other modules that address the key needs for IoT solutions such as security, device management, and reliability.

Build and push a single module

Typically, you'll want to test and debug each module before running it within an entire solution with multiple modules. Because the solution will be build or debug using the Docker engine running inside the EFLOW VM, the first step is building and publishing the module to enable remote debugging.

1. In **Solution Explorer**, select and highlight the module project folder (for example, *myIoTEdgeModule*). Set the custom module as the startup project. Select **Project > Set as StartUp Project** from the menu.
2. To debug the C# Linux module, we need to update *Dockerfile.amd64.debug* file to enable SSH service. Update the *Dockerfile.amd64.debug* file to use the following template: [Dockerfile for Azure IoT Edge AMD64 C# Module with Remote Debug Support ↗](#).

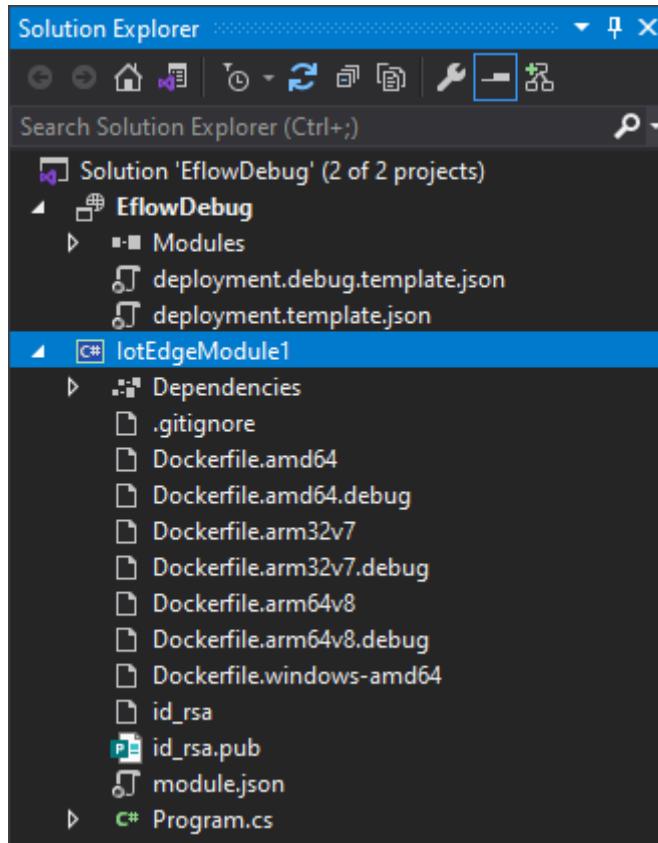
ⓘ Note

When choosing **Debug**, Visual Studio uses `Dockerfile.(amd64|windows-amd64).debug` to build Docker images. This includes the .NET Core command-line debugger VSDBG in your container image while building it. For

production-ready IoT Edge modules, we recommend that you use the **Release** configuration, which uses `Dockerfile.(amd64|windows-amd64)` without VSDBG.

⚠ Warning

Make sure the last line of the template `ENTRYPOINT ["dotnet", "IoTEdgeModule1.dll"]` the name of the DLL matches the name of your IoT Edge module project.



3. To establish an SSH connection with the Linux module, we need to create an RSA key. Open an elevated PowerShell session and run the following commands to create a new RSA key. Make sure you save the RSA key under the same IoT Edge module folder, and the name of the key is `id_rsa`.

Windows Command Prompt

```
ssh-keygen -t RSA -b 4096 -m PEM
```

```
PS C:\Docker> ssh-keygen -t RSA -b 4096 -m PEM
Generating public/private RSA key pair.
Enter file in which to save the key (C:/Users/fcabrera/.ssh/id_rsa): C:/Users/fcabrera/source/repos/EflowDebug/IoTEdgeModule1/id_rsa
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in C:/Users/fcabrera/source/repos/EflowDebug/IoTEdgeModule1/id_rsa.
Your public key has been saved in C:/Users/fcabrera/source/repos/EflowDebug/IoTEdgeModule1/id_rsa.pub.
```

4. If you're using a private registry like Azure Container Registry (ACR), use the following Docker command to sign in to it. You can get the username and password from the **Access keys** page of your registry in the Azure portal. If you're using local registry, you can [run a local registry](#).

```
Windows Command Prompt
```

```
docker -H tcp://<EFLow-VM-IP>:2375 login -u <ACR username> -p <ACR password> <ACR login server>
```

Build module Docker image

Once you've developed your module, you can build the module image to store in a container registry for deployment to your IoT Edge device.

Use the module's Dockerfile to build the module Docker image.

```
Bash
```

```
docker build --rm -f "<DockerFilePath>" -t <ImageNameAndTag> "<ContextPath>"
```

For example, let's assume your command shell is in your project directory and your module name is *IotEdgeModule1*. To build the image for the local registry or an Azure container registry, use the following commands:

```
Bash
```

```
# Build the image for the local registry

docker build --rm -f "./IotEdgeModule1/Dockerfile.amd64.debug" -t
localhost:5000/iotedgemodule1:0.0.1-amd64 "./IotEdgeModule1"

# Or build the image for an Azure Container Registry

docker build --rm -f "./IotEdgeModule1/Dockerfile.amd64.debug" -t
myacr.azurecr.io/iotedgemodule1:0.0.1-amd64 "./IotEdgeModule1"
```

Push module Docker image

Push your module image to the local registry or a container registry.

```
docker push <ImageName>
```

For example:

Bash

```
# Push the Docker image to the local registry

docker push localhost:5000/iotedgemodule1:0.0.1-amd64

# Or push the Docker image to an Azure Container Registry
az acr login --name myacr
docker push myacr.azurecr.io/iotedgemodule1:0.0.1-amd64
```

Deploy the module to the IoT Edge device.

In Visual Studio, open `deployment.debug.template.json` deployment manifest file in the main project. The [deployment manifest](#) is a JSON document that describes the modules to be configured on the targeted IoT Edge device. Before deployment, you need to update your Azure Container Registry credentials, your module images, and the proper `createOptions` values. For more information about `createOption` values, see [How to configure container create options for IoT Edge modules](#).

1. If you're using an Azure Container Registry to store your module image, you need to add your credentials to `deployment.debug.template.json` in the `edgeAgent` settings. For example,

JSON

```
"modulesContent": {
"$edgeAgent": {
"properties.desired": {
"schemaVersion": "1.1",
"runtime": {
"type": "docker",
"settings": {
"minDockerVersion": "v1.25",
"loggingOptions": "",
"registryCredentials": {
"myacr": {
"username": "myacr",
"password": "<your_acr_password>",
"address": "myacr.azurecr.io"
}
}
}
}
},
"//...
```

2. Replace the `image` property value with the module image name you pushed to the registry. For example, if you pushed an image tagged

`myacr.azurecr.io/iotedgemodule1:0.0.1-amd64` for custom module *iotEdgeModule1*, replace the image property value with the tag value.

3. Add or replace the *createOptions* value with stringified content *for each system and custom module in the deployment template*.

For example, the *iotEdgeModule1*'s *image* and *createOptions* settings would be similar to the following:

JSON

```
"IoTEdgeModule1": {  
    "version": "1.0.0",  
    "type": "docker",  
    "status": "running",  
    "restartPolicy": "always",  
    "settings": {  
        "image": "myacr.azurecr.io/iotedgemodule1:0.0.1-amd64",  
        "createOptions": "{\"HostConfig\":{\"PortBindings\":{\"5671/tcp\":[{\"HostPort\":\"5671\"}],\"8883/tcp\":[{\"HostPort\":\"8883\"}],\"443/tcp\":[{\"HostPort\":\"443\"}]}\"}  
    }  
}
```

Use the [IoT Edge Azure CLI set-modules](#) command to deploy the modules to the Azure IoT Hub. For example, to deploy the modules defined in the *deployment.debug.amd64.json* file to IoT Hub *my-iot-hub* for the IoT Edge device *my-device*, use the following command:

Azure CLI

```
az iot edge set-modules --hub-name my-iot-hub --device-id my-device --  
content ./deployment.debug.template.json --login "HostName=my-iot-hub.azure-  
devices.net;SharedAccessKeyName=iothubowner;SharedAccessKey=  
<SharedAccessKey>"
```

Tip

You can find your IoT Hub connection string in the Azure portal under Azure IoT Hub > Security settings > Shared access policies.

1. In **Cloud Explorer**, right-click your edge device and refresh to see the new module running along with **\$edgeAgent** and **\$edgeHub** modules.

Debug the solution

1. Using an elevated PowerShell session run the following commands

- Get the moduleId based on the name used for the Linux C# module. Make sure to replace the <iot-edge-module-name> placeholder with your module's name.

```
PowerShell
```

```
$moduleId = Invoke-EflowVmCommand "sudo docker ps -aqf name=<iot-edge-module-name>"
```

- Check that the \$moduleId is correct - If the variable is empty, make sure you're using the correct module name
- Start the SSH service inside the Linux container

```
PowerShell
```

```
Invoke-EflowVmCommand "sudo docker exec -it -d $moduleId service ssh start"
```

- Open the module SSH port on the EFLOW VM (this tutorial uses port 10022)

```
PowerShell
```

```
Invoke-EflowVmCommand "sudo iptables -A INPUT -p tcp --dport 10022 -j ACCEPT"
```

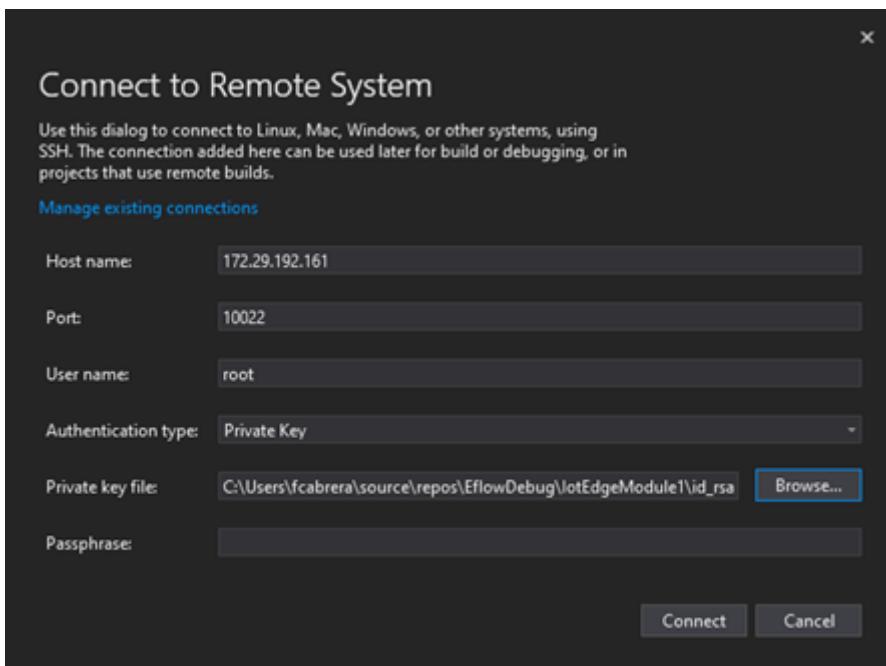
⚠ Warning

For security reasons, every time the EFLOW VM reboots, the IP table rule will delete and go back to the original settings. Also, the module SSH service will have to be started again manually.

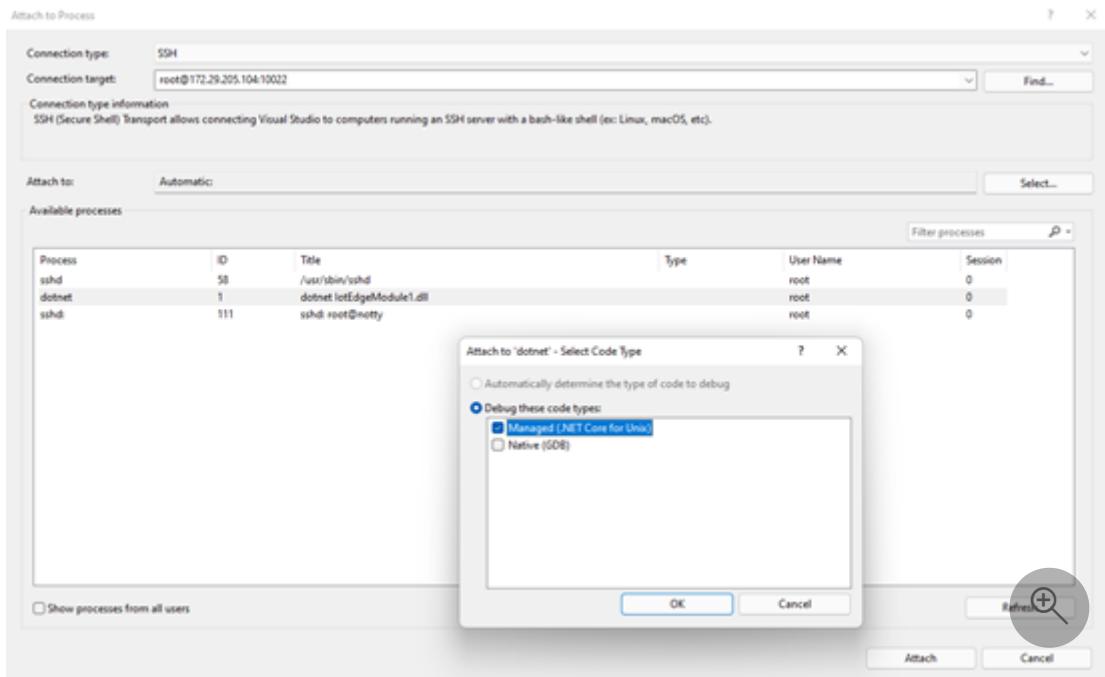
- After successfully starting SSH service, select **Debug -> Attach to Process**, set Connection Type to SSH, and Connection target to the IP address of your EFLOW VM. If you don't know the EFLOW VM IP, you can use the `Get-EflowVmAddr` PowerShell cmdlet. First, type the IP and then press enter. In the pop-up window, input the following configurations:

[+] Expand table

| Field | Value |
|---------------------|--|
| Hostname | Use the EFLOW VM IP |
| Port | 10022 (Or the one you used in your deployment configuration) |
| Username | root |
| Authentication type | Private Key |
| Private Key File | Full path to the id_rsa that created in a previous step |
| Passphrase | Passphrase used for the key created in a previous step |



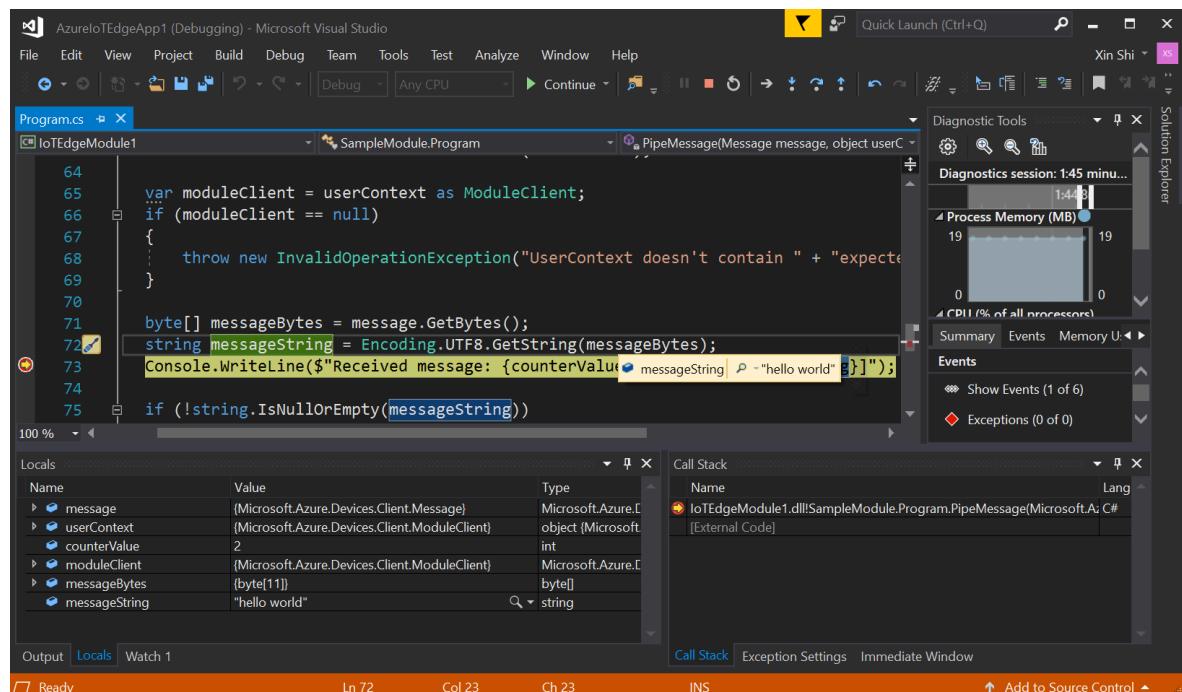
- After successfully connecting to the module using SSH, then you can choose the process and select Attach. For the C# module you need to choose process dotnet and **Attach to** to Managed (CoreCLR). It may take 10 to 20 seconds the first time.



4. Set a breakpoint to inspect the module.

- If developing in C#, set a breakpoint in the `PipeMessage()` function in **ModuleBackgroundService.cs**.
- If using C, set a breakpoint in the `InputQueue1Callback()` function in **main.c**.

5. The output of the **SimulatedTemperatureSensor** should be redirected to **input1** of the custom Linux C# module. The breakpoint should be triggered. You can watch variables in the Visual Studio **Locals** window.



6. Press **Ctrl + F5** or select the stop button to stop debugging.

Clean up resources

If you plan to continue to the next recommended article, you can keep the resources and configurations that you created and reuse them. You can also keep using the same IoT Edge device as a test device.

Otherwise, you can delete the local configurations and the Azure resources that you used in this article to avoid charges.

Delete Azure resources

Deleting Azure resources and resource groups is irreversible. Make sure that you don't accidentally delete the wrong resource group or resources. If you created the IoT hub inside an existing resource group that has resources that you want to keep, delete only the IoT hub resource itself, not the resource group.

To delete the resources:

1. Sign in to the [Azure portal](#), and then select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources that are contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can click into each resource to delete them individually.

Next steps

In this tutorial, you set up Visual Studio on your development machine and deployed and debugged your first IoT Edge module from it. Now that you know the basic concepts, try adding functionality to a module so that it can analyze the data passing through it.

[Develop Azure IoT Edge modules using Visual Studio Code](#)

Tutorial: Create a hierarchy of IoT Edge devices using IoT Edge for Linux on Windows

Article • 06/10/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

You can deploy Azure IoT Edge nodes across networks organized in hierarchical layers. Each layer in a hierarchy is a gateway device that handles messages and requests from devices in the layer beneath it. This configuration is also known as *nested edge*.

You can structure a hierarchy of devices so that only the top layer has connectivity to the cloud, and the lower layers can only communicate with adjacent north and south layers. This network layering is the foundation of most industrial networks, which follow the [ISA-95 standard](#).

This tutorial walks you through creating a hierarchy of IoT Edge devices using IoT Edge for Linux on Windows, deploying IoT Edge runtime containers to your devices, and configuring your devices locally. You do the following tasks:

- ✓ Create and define the relationships in a hierarchy of IoT Edge devices.
- ✓ Configure the IoT Edge runtime on the devices in your hierarchy.
- ✓ Install consistent certificates across your device hierarchy.
- ✓ Add workloads to the devices in your hierarchy.
- ✓ Use the [IoT Edge API Proxy module](#) to securely route HTTP traffic over a single port from your lower layer devices.

Tip

This tutorial includes a mixture of manual and automated steps to provide a showcase of nested IoT Edge features.

If you would like an entirely automated look at setting up a hierarchy of IoT Edge devices, you guide your own script based on the scripted [Azure IoT Edge for](#)

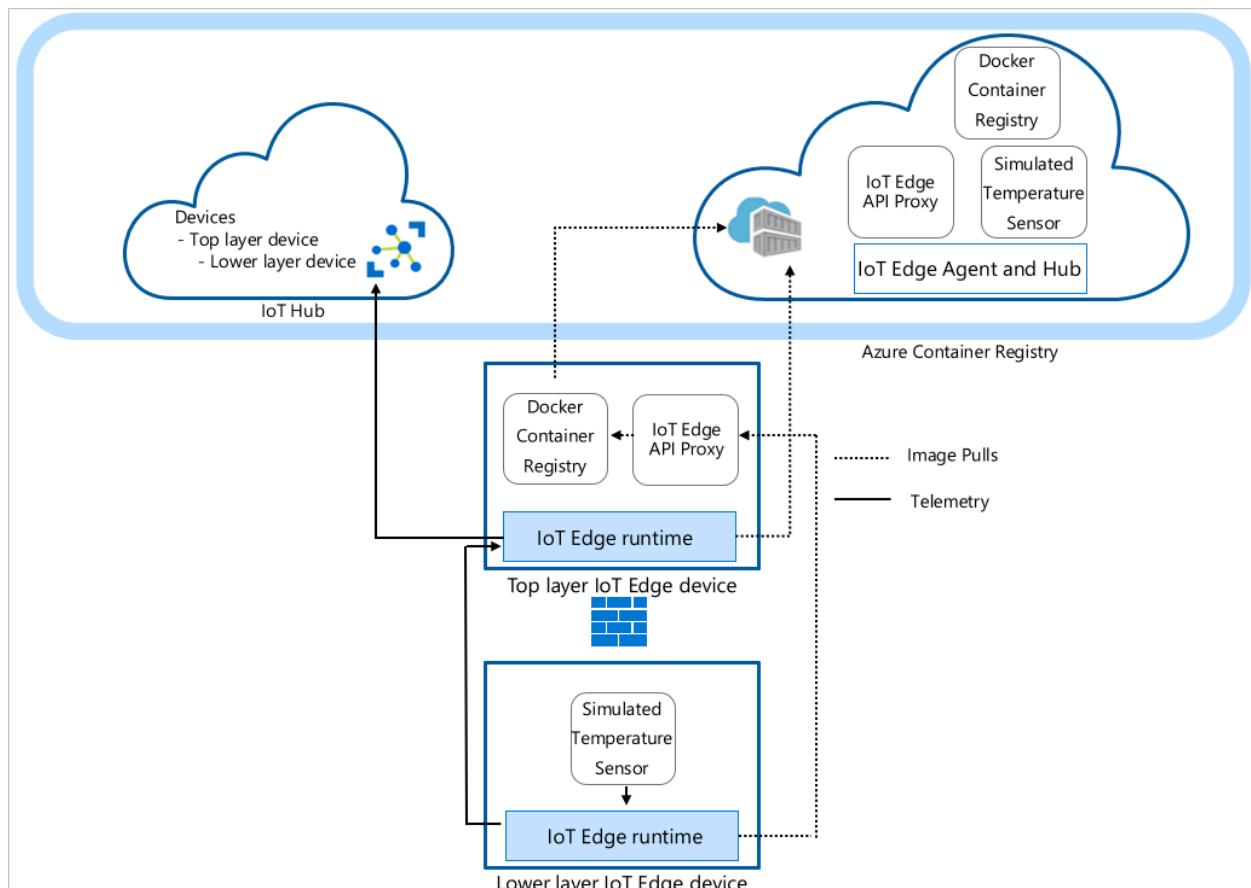
[Industrial IoT sample](#). This scripted scenario deploys Azure virtual machines as preconfigured devices to simulate a factory environment.

If you would like an in-depth look at the manual steps to create and manage a hierarchy of IoT Edge devices, see [the how-to guide on IoT Edge device gateway hierarchies](#).

In this tutorial, the following network layers are defined:

- **Top layer:** IoT Edge devices at this layer can connect directly to the cloud.
- **Lower layers:** IoT Edge devices at layers below the top layer can't connect directly to the cloud. They need to go through one or more intermediary IoT Edge devices to send and receive data.

This tutorial uses a two device hierarchy for simplicity. The **top layer device** represents a device at the top layer of the hierarchy that can connect directly to the cloud. This device is referred to as the **parent device**. The **lower layer device** represents a device at the lower layer of the hierarchy that can't connect directly to the cloud. You can add more devices to represent your production environment, as needed. Devices at lower layers are referred to as **child devices**.



① Note

A child device can be a downstream device or a gateway device in a nested topology.

Prerequisites

To create a hierarchy of IoT Edge devices, you need:

- A Bash shell in Azure Cloud Shell using Azure CLI v2.3.1 with the Azure IoT extension v0.10.6 or higher installed. This tutorial uses the [Azure Cloud Shell](#). To see your current versions of the Azure CLI modules and extensions, run `az version`.
- Two Windows devices running Azure IoT Edge for Linux on Windows. Both devices should be deployed using an [external virtual switch](#).

💡 Tip

It is possible to use **internal** or **default** virtual switch if a port forwarding is configured on the Windows host OS. However, for the simplicity of this tutorial, both devices should use an **external** virtual switch and be connected to the same external network.

For more information about networking, see [Azure IoT Edge for Linux on Windows networking](#) and [Networking configuration for Azure IoT Edge for Linux on Windows](#).

If you need to set up the EFLOW devices on a DMZ, see [How to configure Azure IoT Edge for Linux on Windows Industrial IoT & DMZ configuration](#).

- An Azure account with a valid subscription. If you don't have an [Azure subscription](#), create a [free account](#) before you begin.
- A free or standard tier [IoT Hub](#) in Azure.
- Make sure that the following ports are open inbound for all devices except the lowest layer device: 443, 5671, 8883:
 - 443: Used between parent and child edge hubs for REST API calls and to pull docker container images.
 - 5671, 8883: Used for AMQP and MQTT.

💡 Tip

For more information on EFLOW virtual machine firewall, see [IoT Edge for Linux on Windows security](#).

Create your IoT Edge device hierarchy

IoT Edge devices make up the layers of your hierarchy. This tutorial creates a hierarchy of two IoT Edge devices: the *top layer device* and the *lower layer device*. You can create more downstream devices as needed.

To create and configure your hierarchy of IoT Edge devices, you use the [az iot edge devices create](#) Azure CLI command. The command simplifies the configuration of the hierarchy by automating and condensing several steps:

- Creates devices in your IoT Hub
- Sets the parent-child relationships to authorize communication between devices
- Applies the deployment manifest to each device
- Generates a chain of certificates for each device to establish secure communication between them
- Generates configuration files for each device

Create device configuration

You create a group of nested edge devices with containing a parent device with one child device. In this tutorial, we use basic sample deployment manifests. For other scenario examples, review the [configuration example templates](#).

1. Before you use the [az iot edge devices create](#) command, you need to define the deployment manifest for the top layer and lower layer devices. Download the [deploymentTopLayer.json](#) sample file to your local machine.

The top layer device deployment manifest defines the [IoT Edge API Proxy](#) module and declares the [route](#) from the lower layer device to IoT Hub.

2. Download the [deploymentLowerLayer.json](#) sample file to your local machine.

The lower layer device deployment manifest includes the simulated temperature sensor module and declares the [route](#) to the top layer device. You can see within **systemModules** section that the runtime modules are set to pull from `$upstream:443`, instead of `mcr.microsoft.com`. The *lower layer device* sends Docker image requests the *IoT Edge API Proxy* module on port 443, as it can't directly pull the images from the cloud. The other module deployed to the *lower layer device*, the *Simulated Temperature Sensor* module, also makes its image request to `$upstream:443`.

For more information on how to create a lower layer deployment manifest, see [Connect Azure IoT Edge devices to create a hierarchy](#).

3. In the Azure Cloud Shell [↗](#), use the `az iot edge devices create` Azure CLI command to create devices in IoT Hub and configuration bundles for each device in your hierarchy. Replace the following placeholders with the appropriate values:

[+] Expand table

| Placeholder | Description |
|---|---|
| <code><hub-name></code> | The name of your IoT Hub. |
| <code><config-bundle-output-path></code> | The folder path where you want to save the configuration bundles. |
| <code><parent-device-name></code> | The <i>top layer</i> parent device ID name. |
| <code><parent-deployment-manifest></code> | The parent device deployment manifest file. |
| <code><parent-fqdn-or-ip></code> | Parent device fully qualified domain name (FQDN) or IP address. |
| <code><child-device-name></code> | The <i>lower layer</i> child device ID name. |
| <code><child-deployment-manifest></code> | The child device deployment manifest file. |
| <code><child-fqdn-or-ip></code> | Child device fully qualified domain name (FQDN) or IP address. |

Azure CLI

```
az iot edge devices create \
  --hub-name <hub-name> \
  --output-path <config-bundle-output-path> \
  --default-edge-agent "mcr.microsoft.com/azureiotedge-agent:1.4" \
  --device id=<parent-device-name> \
    deployment=<parent-deployment-manifest> \
    hostname=<parent-fqdn-or-ip> \
  --device id=child-1 \
    parent=parent-1 \
    deployment=<child-deployment-manifest> \
    hostname=<child-fqdn-or-ip>
```

For example, the following command creates a hierarchy of two IoT Edge devices in IoT Hub. A top layer device named *parent-1* and a lower layer device named *child-1**. The command saves the configuration bundles for each device in the *output* directory. The command also generates self-signed test certificates and includes them in the configuration bundle. The configuration bundles are installed on each device using an install script.

Azure CLI

```
az iot edge devices create \
--hub-name my-iot-hub \
--output-path ./output \
--default-edge-agent "mcr.microsoft.com/azureiotedge-agent:1.4" \
--device id=parent-1 \
    deployment=./deploymentTopLayer.json \
    hostname=10.0.0.4 \
--device id=child-1 \
    parent=parent-1 \
    deployment=./deploymentLowerLayer.json \
    hostname=10.1.0.4
```

After running the command, you can find the device configuration bundles in the output directory. For example:

Output

```
PS C:\nested-edge\output> dir

Directory: C:\nested-edge\output

Mode                LastWriteTime         Length  Name
----                -----          -----  --
-a---        4/10/2023  4:12 PM           7192  child-1.tgz
-a---        4/10/2023  4:12 PM          6851  parent-1.tgz
```

You can use your own certificates and keys passed as arguments to the command or create a more complex device hierarchy. For more information about creating nested devices using the `az` command, see [az iot edge devices create](#). If you're unfamiliar with how certificates are used in a gateway scenario, see the how-to guide's [certificate section](#).

In this tutorial, you use inline arguments to create the devices and configuration bundles. You can also use a configuration file in YAML or JSON format. For a sample configuration file, see the example [sample_devices_config.yaml](#).

Configure the IoT Edge runtime

In addition to the provisioning of your devices, the configuration steps establish trusted communication between the devices in your hierarchy using the certificates you created earlier. The steps also begin to establish the network structure of your hierarchy. The top layer device maintains internet connectivity, allowing it to pull images for its runtime from the cloud, while lower layer devices route through the top layer device to access these images.

To configure the IoT Edge runtime, you need to apply the configuration bundles to your devices. The configurations differ between the *top layer device* and a *lower layer device*, so be mindful of the device configuration file you're applying to each device.

Each device needs its corresponding configuration bundle. You can use a USB drive or [secure file copy](#) to move the configuration bundles to each device. You need to copy the configuration bundle to the Windows host OS of each EFLOW device and then copy it to the EFLOW VM.

⚠️ Warning

Be sure to send the correct configuration bundle to each device.

Top-layer device configuration

1. Connect to your *top level* Windows host device and copy the **parent-1.tzg** file to the device.
2. Start an elevated *PowerShell* session using **Run as Administrator**.
3. Copy **parent-1.tzg** into the EFLOW VM.

PowerShell

```
Copy-EflowVmFile -fromFile parent-1.tzg -toFile ~/ -pushFile
```

4. Connect to your EFLOW virtual machine

PowerShell

```
Connect-EflowVm
```

5. Extract the configuration bundle archive. For example, use the *tar* command to extract the *parent-1* archive file:

Bash

```
tar -xzf ./parent-1.tgz
```

6. Set execute permission for the install script.

Bash

```
chmod +x install.sh
```

7. Run the `install.sh` script.

Bash

```
sudo sh ./install.sh
```

8. Apply the correct certificate permissions and restart the IoT Edge runtime.

Bash

```
sudo chmod -R 755 /etc/aziot/certificates/
sudo iotedge system restart
```

9. Check that all IoT Edge services are running correctly.

Bash

```
sudo iotedge system status
```

10. Finally, add the appropriate firewall rules to enable connectivity between the lower-layer device and top-layer device.

Bash

```
sudo iptables -A INPUT -p tcp --dport 5671 -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 8883 -j ACCEPT
sudo iptables -A INPUT -p tcp --dport 443 -j ACCEPT
sudo iptables -A INPUT -p icmp --icmp-type 8 -s 0/0 -m state --state NEW,ESTABLISHED,RELATED -j ACCEPT
sudo iptables-save | sudo tee /etc/systemd/scripts/ip4save
```

11. Run the configuration and connectivity checks on your devices.

Bash

```
sudo iotedge check
```

On your **top layer device**, expect to see an output with several passing evaluations. You may see some warnings about logs policies and, depending on your network, DNS policies.

If you want a closer look at what modifications are being made to your device's configuration file, see [the configure IoT Edge on devices section of the how-to guide](#).

Lower-layer device configuration

1. Connect to your *lower level* Windows host device and copy the **child-1.tgz** file to the device.
2. Start an elevated *PowerShell* session using **Run as Administrator**.
3. Copy **child-1.tgz** into the EFLOW VM.

```
PowerShell
```

```
Copy-EflowVmFile -fromFile child-1.tgz -toFile ~/ -pushFile
```

4. Connect to your EFLOW virtual machine

```
PowerShell
```

```
Connect-EflowVm
```

5. Extract the configuration bundle archive. For example, use the *tar* command to extract the *child-1* archive file:

```
Bash
```

```
tar -xzf ./child-1.tgz
```

6. Set execute permission for the install script.

```
Bash
```

```
chmod +x install.sh
```

7. Run the **install.sh** script.

```
Bash
```

```
sudo sh ./install.sh
```

8. Apply the correct certificate permissions and restart the IoT Edge runtime.

```
Bash
```

```
sudo chmod -R 755 /etc/aziot/certificates/  
sudo iotedge system restart
```

9. Check that all IoT Edge services are running correctly.

```
Bash
```

```
sudo iotedge system status
```

10. Run the configuration and connectivity checks on your devices. For the **lower layer device**, the diagnostics image needs to be manually passed in the command:

```
Bash
```

```
sudo iotedge check --diagnostics-image-name  
<parent_device_fqdn_or_ip>:443/azureiotedge-diagnostics:1.2
```

If you completed the earlier steps correctly, you can verify your devices are configured correctly. Once you're satisfied your configurations are correct on each device, you're ready to proceed.

Device module deployment

The module deployment for your devices were applied when the devices were created in IoT Hub. The *az iot edge devices create* command applied the deployment JSON files for the top and lower layer devices. After those deployments completed, the **lower layer device** uses the **IoT Edge API Proxy** module to pull its necessary images.

In addition the runtime modules **IoT Edge Agent** and **IoT Edge Hub**, the **top layer device** receives the **Docker registry** module and **IoT Edge API Proxy** module.

The **Docker registry** module points to an existing Azure Container Registry. In this case, `REGISTRY_PROXY_REMOTEURL` points to the Microsoft Container Registry. By default, **Docker registry** listens on port 5000.

The *IoT Edge API Proxy* module routes HTTP requests to other modules, allowing lower layer devices to pull container images or push blobs to storage. In this tutorial, it communicates on port 443 and is configured to send Docker container image pull requests route to your **Docker registry** module on port 5000. Also, any blob storage upload requests route to module `AzureBlobStorageonIoTEdge` on port 11002. For more

information about the IoT Edge API Proxy module and how to configure it, see the module's [how-to guide](#).

If you'd like a look at how to create a deployment like this through the Azure portal or Azure Cloud Shell, see [top layer device section of the how-to guide](#).

You can view the status of your modules using the command:

Azure CLI

```
az iot hub module-twin show --device-id <edge-device-id> --module-id  
'$edgeAgent' --hub-name <iot-hub-name> --query "properties.reported.  
[systemModules, modules]"
```

This command outputs all the edgeAgent reported properties. Here are some helpful ones for monitoring the status of the device: *runtime status*, *runtime start time*, *runtime last exit time*, *runtime restart count*.

You can also see the status of your modules on the [Azure portal](#). Navigate to the **Devices** section of your IoT Hub to see your devices and modules.

View generated data

The **Simulated Temperature Sensor** module that you pushed generates sample environment data. It sends messages that include ambient temperature and humidity, machine temperature and pressure, and a timestamp.

You can also view these messages through the [Azure Cloud Shell](#):

Azure CLI

```
az iot hub monitor-events -n <iot-hub-name> -d <lower-layer-device-name>
```

For example:

Azure CLI

```
az iot hub monitor-events -n my-iot-hub -d child-1
```

Output

```
{  
  "event": {  
    "origin": "child-1",  
    "temperature": 23.5,  
    "humidity": 45.2,  
    "machineTemp": 50.1,  
    "machinePressure": 101.3,  
    "timestamp": "2023-10-01T12:00:00Z"  
  }  
}
```

```
        "module": "simulatedTemperatureSensor",
        "interface": "",
        "component": "",
        "payload": "{\"machine\":
{\"temperature\":104.29281270901808,\"pressure\":10.48905461241978},\"ambien
t\":
{\"temperature\":21.086561171611102,\"humidity\":24},\"timeCreated\":\"2023-
04-17T21:50:30.1082487Z\"}"
    }
}
```

Troubleshooting

Run the `iotedge check` command to verify the configuration and to troubleshoot errors.

You can run `iotedge check` in a nested hierarchy, even if the downstream devices don't have direct internet access.

When you run `iotedge check` from the lower layer, the program tries to pull the image from the parent through port 443.

Bash

```
sudo iotedge check --diagnostics-image-name $upstream:443/azureiotedge-
diagnostics:1.2
```

The `azureiotedge-diagnostics` value is pulled from the container registry that's linked with the registry module. This tutorial has it set by default to <https://mcr.microsoft.com>:

[+] [Expand table](#)

| Name | Value |
|--------------------------|---|
| REGISTRY_PROXY_REMOTEURL | https://mcr.microsoft.com |

If you're using a private container registry, make sure that all the images (IoTEdgeAPIProxy, edgeAgent, edgeHub, Simulated Temperature Sensor, and diagnostics) are present in the container registry.

If a downstream device has a different processor architecture from the parent device, you need the appropriate architecture image. You can use a [connected registry](#) or you can specify the correct image for the `edgeAgent` and `edgeHub` modules in the downstream device `config.toml` file. For example, if the parent device is running on an

ARM32v7 architecture and the downstream device is running on an AMD64 architecture, you need to specify the matching version and architecture image tag in the downstream device *config.toml* file.

```
toml

[agent.config]
image = "$upstream:443/azureiotedge-agent:1.4.10-linux-amd64"

"systemModules": {
    "edgeAgent": {
        "settings": {
            "image": "$upstream:443/azureiotedge-agent:1.4.10-linux-amd64"
        },
    },
    "edgeHub": {
        "settings": {
            "image": "$upstream:443/azureiotedge-hub:1.4.10-linux-amd64",
        }
    }
}
```

Clean up resources

You can delete the local configurations and the Azure resources that you created in this article to avoid charges.

To delete the resources:

1. Sign in to the [Azure portal](#) and select **Resource groups**.
2. Select the name of the resource group that contains your IoT Edge test resources.
3. Review the list of resources contained in your resource group. If you want to delete all of them, you can select **Delete resource group**. If you want to delete only some of them, you can select each resource to delete them individually.

Next steps

In this tutorial, you configured two IoT Edge devices as gateways and set one as the parent device of the other. Then, you demonstrated pulling a container image onto the child device through a gateway using the IoT Edge API Proxy module. See [the how-to guide on the proxy module's use](#) if you want to learn more.

To learn more about using gateways to create hierarchical layers of IoT Edge devices, see the following article.

[Connect Azure IoT Edge devices to create a hierarchy](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Troubleshoot your IoT Edge for Linux on Windows device

Article • 06/10/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

If you experience issues running Azure IoT Edge for Linux on Windows (EFLOW) in your environment, use this article as a guide for troubleshooting and diagnostics.

You can also check [IoT Edge for Linux on Windows GitHub issues](#) for a similar reported issue.

Isolate the issue

Your first step when troubleshooting IoT Edge for Linux on Windows should be to understand which component is causing the issue. There are three main components an EFLOW solution:

- Windows components: PowerShell module, WSSDAgent & EFLOWProxy
- CBL Mariner Linux virtual machine
- Azure IoT Edge

For more information about EFLOW architecture, see [What is Azure IoT Edge for Linux on Windows](#).

If your issue is installing or deploying the EFLOW virtual machine, make sure that all the prerequisites are met, and verify your networking and VM configurations. If your installation and deployment was successful and you're facing issues with the post VM management, the problems are generally related to VM lifecycle, networking, or Azure IoT Edge. Finally, if the issue is related to modules or IoT Edge features, check [Troubleshoot your IoT Edge device](#).

For more information about common errors related to *installation and deployment*, *provisioning*, *interaction with the VM*, and *networking*, see [Common issues and resolutions for Azure IoT Edge for Linux on Windows](#).

Gather debug information

When you need to gather logs from an IoT Edge for Linux on Windows device, the most convenient way is to use the `Get-EflowLogs` PowerShell cmdlet. By default, this command collects the following logs:

- **eflowlogs-summary.txt**: contains the status of all log collection steps.
- **EFLOW VM configuration**: includes the VM, networking, and passthrough configurations and additional information.
- **EFLOW Events** : Windows events related to the VM lifecycle and *EFLOWProxy* service.
- **IoT Edge logs**: includes the output of `iotedge check` the IoT Edge runtime support bundle.
- **WSSDAgent logs**: includes all the logs related to the *WSSDAgent* service.

After the cmdlet gathers all the required logs, the files are compressed into a single file named *eflowlogs.zip* under the EFLOW installation path (For example, *C:\Program Files\Azure IoT Edge*).

Check your IoT Edge version

If you're running an older version of IoT Edge for Linux on Windows, then upgrading may resolve your issue. To check the EFLOW version installed on your device, use the following steps:

1. Open **Settings** on Windows.
2. Select **Add or Remove Programs**.
3. Depending on the EFLOW release train being used (Continuous Release or LTS), choose **Azure IoT Edge LTS** or **Azure IoT Edge**.
4. Check the version under the EFLOW app name.

For more information about specific versions release notes, check [Azure IoT Edge for Linux on Windows release notes](#).

For instructions on how to update your device, see [Update IoT Edge for Linux on Windows](#).

Check the EFLOW VM status

You can verify the EFLOW VM status and information by using the `Get-EflowVm` PowerShell cmdlet. If the EFLOW VM is running, the **VmPowerState** output should be

Running. Whereas if the VM is stopped, the `VmPowerState` output is *Off*. To start or stop the EFLOW VM, use the `Start-EflowVm` and `Stop-EflowVm` cmdlet.

If the VM is *Running* but you can't interact or access the VM, there's probably a networking issue between the VM and the Windows host OS. Also, make sure that the EFLOW VM has enough memory and storage available to continue with normal execution. Run the `Get-EflowVm` cmdlet to see the memory(*TotalMemMb*, *UsedMemMb*, *AvailableMemMb*) and storage(*TotalStorageMb*, *UsedStorageMb*, *AvailableStorageMb*) information.

Finally, if the VM is *Off* and you can't start it using the `Start-EflowVm` cmdlet, there may be several reasons why the VM can't be started.

First, the issue could be related to the VM lifecycle management service (*WSSDAgent*) not running. Ensure that the *WSSDAgent* service is running using the following steps:

1. Start an elevated *PowerShell* session using **Run as Administrator**.
2. Check the service status

```
PowerShell  
  
Get-Service -Name WSSDAgent
```

3. If the service is **Stopped**, start the service using the following command:

```
PowerShell  
  
Start-Service -Name WSSDAgent
```

4. If the service is **Running**, the issue is probably related to a networking misconfiguration or lack of resources to create the VM.

Second, the issue could be related to lack of resources. You can set the *EflowVmAssignedMemory* (-*memoryInMb*) and *EflowVmAssignedCPUCores* (-*cpuCount*) assigned to the VM during deployment using the `Deploy-Eflow` PowerShell cmdlet, or after deployment using the `Set-EflowVm` cmdlet. If these resources aren't available when trying to start the VM, the VM fails to start. To check the resources assigned and available, use the following steps:

1. Start an elevated *PowerShell* session using **Run as Administrator**.
2. Check the available memory. Ensure that the *FreePhysicalMemory* is greater than the *EflowVmAssignedMemory*.

```
PowerShell
```

```
Get-CIMInstance Win32_OperatingSystem | Select FreePhysicalMemory
```

3. Check the available CPU cores. Ensure that *NumberOfLogicalProcessors* is greater than *EflowVmAssignedCPUCores*.

PowerShell

```
wmic cpu get NumberOfLogicalProcessors
```

Finally, the issue could be related to networking. For more information about EFLOW VM networking issues, see [How to troubleshoot Azure IoT Edge for Linux on Windows networking](#).

Check the status of the IoT Edge runtime

The [IoT Edge runtime](#) is responsible for receiving the code to run at the edge and communicate the results. If IoT Edge runtime and modules aren't running, no code runs at the edge. You can check the runtime and module status using the following steps:

1. Start an elevated *PowerShell* session using [Run as Administrator](#).
2. Check the IoT Edge runtime status. In particular, check if the service is **Loaded** and **Active**.

PowerShell

```
(Get-EflowVm).EdgeRuntimeStatus.SystemCtlStatus | Format-List
```

3. Check the IoT Edge module status. Check that all modules are running.

PowerShell

```
(Get-EflowVm).EdgeRuntimeStatus.ModuleList | Format-List
```

For more information about IoT Edge runtime troubleshooting, see [Troubleshoot your IoT Edge device](#).

Check TPM passthrough

If you're using TPM provisioning by following the guide [Create and provision an IoT Edge for Linux on Windows device at scale by using a TPM](#), you must enable TPM passthrough. In order to access the physical TPM connected to the Windows host OS, all the EFLOW VM TPM commands are forwarded to the host OS using a Windows service

called *EFLWProxy*. If you experience issues using *DpsTpm* provisioning, or accessing TPM indexes from the EFLW VM, check the service status using the following steps:

1. Start an elevated *PowerShell* session using **Run as Administrator**.
2. Check the status of the *EFLWProxy* service.

```
PowerShell
```

```
Get-Service -Name EFLWProxy
```

3. If the service is **Stopped**, start the service using the following command:

```
PowerShell
```

```
Start-Service -Name EFLWProxy
```

If the service won't start, check the *EFLWProxy* logs. Go to **Apps > Event Viewer > Applications and Services Logs > Microsoft > EFLW > EFLWProxy** and check the logs.

4. If the service is **Running** then check the EFLW VM proxy services. Start by connecting to the EFLW VM.

```
PowerShell
```

```
Connect-EflowVm
```

5. From inside the EFLW VM, check the TPM services are up and running.

```
Bash
```

```
sudo systemctl status tpm*
```

You should see the status and logs of four different services. The four services should be up and running.

- a. **tpm2-netns.service** - TPM2 Network Namespace
- b. **tpm2-socat@2322.service** - TPM2 Sandbox Service on Port 2322
- c. **tpm2-socat@2321.service** - TPM2 Sandbox Service on Port 2321
- d. **tpm2-abrmd.service** - TPM2 Access Broker and Resource Management

Daemon

If any of these services is **stopped** or **failed**, restart all services using the following command:

```
Bash
```

```
sudo systemctl restart tpm*
```

6. Check the communication between the EFLOW VM and the *EFLOWProxy* service. If communication is working, you should see the *RegistrationId* and the TPM *Endorsement Key* as output from the following command:

```
Bash
```

```
sudo /usr/bin/tpm_device_provision
```

Check GPU Assignment

If you're using GPU passthrough, ensure to follow all the prerequisites and configurations outlined in [GPU acceleration for Azure IoT Edge for Linux on Windows](#). If you experience issues using GPU passthrough feature, check the following steps:

First, start by checking your device is available on the Windows host OS.

1. Open **Apps > Device Manager**.
2. Go to **Display Adapters** and check that your GPU is in the list.
3. Right-click the GPU name and select **Properties**.
4. Check that the driver is correctly installed.

Second, if the GPU is correctly assigned, but still not being able to use it inside the EFLOW VM, use the following steps:

1. Start an elevated *PowerShell* session using **Run as Administrator**.
2. Connect to the EFLOW VM

```
PowerShell
```

```
Connect-EflowVm
```

3. If you're using a **NVIDIA GPU**, check the passthrough status using the following command:

```
Bash
```

```
sudo nvidia-smi
```

You should be able to see the GPU card information, driver version, CUDA version, and the GPU system and processes information.

4. If you're using an Intel iGPU passthrough, check the passthrough status using the following command:

```
Bash
```

```
sudo ls -al /dev/dxg
```

The expected output should be similar to:

```
Output
```

```
crw-rw-rw- 1 root 10, 60 Sep 8 06:20 /dev/dxg
```

For more Intel iGPU performance and debugging information, see [Witness the power of Intel® iGPU with Azure IoT Edge for Linux on Windows\(EFLOW\) & OpenVINO™ Toolkit](#).

Check WSSDAgent logs for issues

The first step before checking *WSSDAgent* logs is to check if the VM was created and is running.

1. Start an elevated *PowerShell* session using **Run as Administrator**.
2. On Windows Client SKUs, check the [HCS](#) virtual machines.

```
PowerShell
```

```
hcsdiag list
```

If the EFLOW VM is running, you should see a line that contains a GUID followed by *wssdagent*. For example:

```
Output
```

```
2bd841e4-126a-11ed-9a91-f01dbca16d1e
VM, Running, 2BD841E4-126A-11ED-9A91-
F01DBCA16D1E, wssdagent
88d7aa8c-0d1f-4786-b4cb-62eff1decd92
```

VM,
B4CB-62EFF1DECD92, CmService

SavedAsTemplate, 88D7AA8C-0D1F-4786-

3. On Windows Server SKUs, check the [VMMS](#) virtual machines

PowerShell

```
hcsdiag list
```

If the EFLOW VM is running, you should see a line that contains the <WindowsHostname-EFLOW> as a name. For example:

Output

| Name | State | CPUUsage(%) | MemoryAssigned(M) | Uptime |
|------------------|--------------------|-------------|-------------------|--------|
| Status | Version | | | |
| ---- | ----- | ----- | ----- | ----- |
| NUC-EFLOW | Running | 0 | 1024 | |
| 00:01:34.1280000 | Operating normally | 9.0 | | |

If for some reason the VM isn't listed, that means that VM isn't running or the *WSSDAgent* wasn't able to create it. Use the following steps to check the *WSSDAgent* logs:

1. Open **File Explorer**.
2. Go to `C:\ProgramData\wssdagent\log`
3. Open the `wssdagent.log` file.
4. Look for the words **Error** or **Fail**.

Reinstall EFLOW

Sometimes, a system might require significant special modification to work with existing networking or operating system constraints. For example, a system could require complex networking configurations (firewall, Windows policies, proxy settings) and custom Windows OS configurations. If you tried all previous troubleshooting steps and still have EFLOW issues, it's possible that there's some misconfiguration that is causing the issue. In this case, the final option is to uninstall and reinstall EFLOW.

Next steps

Do you think that you found a bug in the IoT Edge for Linux on Windows? [Submit an issue](#) so that we can continue to improve.

If you have more questions, create a [Support request](#) for help.

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback](#)

Troubleshoot your IoT Edge for Linux on Windows networking

Article • 06/10/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

If you experience networking issues using Azure IoT Edge for Linux on Windows (EFLOW) in your environment, use this article as a guide for troubleshooting and diagnostics. Also, check [Troubleshoot your IoT Edge for Linux on Windows device](#) for more EFLOW virtual machine troubleshooting help.

Isolate the issue

When troubleshooting IoT Edge for Linux on Windows networking, there are four network features that could be causing issues:

- IP addresses configuration
- Domain Name System (DNS) configuration
- Firewall and port configurations
- Other components

For more information about EFLOW networking concepts, see [IoT Edge for Linux on Windows networking](#). Also, for more information about EFLOW networking configurations, see [Networking configuration for Azure IoT Edge for Linux on Windows](#).

Check IP addresses

Your first step when troubleshooting IoT Edge for Linux on Windows networking should be to check the VM IP address configurations. If IP communication is misconfigured, then all inbound and outbound connections fail.

1. Start an elevated *PowerShell* session using [Run as Administrator](#).
2. Check the IP address returned by the VM lifecycle agent. Make note of the IP address and compare it with the one obtained from inside the VM in the later

steps.

PowerShell

[Get-EflowVmAddr](#)

3. Connect to the EFLOW virtual machine.

PowerShell

[Connect-EflowVm](#)

4. Check the *eth0* VM network interface configuration.

Bash

`ifconfig eth0`

In the output, you should see the *eth0* configuration information. Ensure that the *inet* address is correctly set.

Output

```
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
      inet 172.31.100.171 netmask 255.255.240.0 broadcast
            172.31.111.255
              inet6 fe80::215:5dff:fe2a:2f62 prefixlen 64 scopeid 0x20<link>
                ether 00:15:5d:2a:2f:62 txqueuelen 1000 (Ethernet)
                  RX packets 115746 bytes 11579209 (11.0 MiB)
                  RX errors 0 dropped 0 overruns 0 frame 0
                  TX packets 976 bytes 154184 (150.5 KiB)
                  TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

If the *inet* IP address is blank or different from the one provided by using the [Get-EflowVmAddr](#) cmdlet, you need to troubleshoot why the EFLOW VM has an invalid or no IP address assigned. Use the following table to troubleshoot the issue:

[+] [Expand table](#)

| Virtual Switch | IP Address Assignment | Troubleshoot |
|----------------|-----------------------|--|
| External | Static IP | Ensure that the IP configuration is correctly set up. All three parameters <i>ip4Address</i> , <i>ip4GatewayAddress</i> , and <i>ip4PrefixLength</i> should be used. The IP address assigned to the VM should be valid and not being used by other device on the external network. You can |

| Virtual Switch | IP Address Assignment | Troubleshoot |
|----------------|-----------------------|---|
| | | check EFLOW VM interface configurations by checking the files under <code>/etc/systemd/network/</code> . |
| External | DHCP | Ensure that there's a DHCP server on the external network. If no DHCP server is available, then use static IP configurations. Also, make sure that DHCP server has no firewall policy regarding MAC addresses. If it has, you can get the EFLOW MAC address by using the <code>Get-EflowVmAddr</code> cmdlet. |
| Default switch | DHCP | Generally, the issue is related to a malfunction of the default switch. Try rebooting the Windows host OS. If the problem persists, try disabling and enabling Hyper-V |
| Internal | Static IP | Ensure that the IP configuration is correctly set up. All three parameters <code>ip4Address</code> , <code>ip4GatewayAddress</code> , and <code>ip4PrefixLength</code> should be used. The IP address assigned to the VM should be valid and not being used by other device on the internal network. Also, to get connected to internet, you'll need to set up a NAT table. Follow the NAT configuration steps in Azure IoT Edge for Linux on Windows virtual switch creation . |
| Internal | DHCP | Ensure that there's a DHCP server on the internal network. To set up a DHCP server and a NAT table on Windows Server, follow the steps in Azure IoT Edge for Linux on Windows virtual switch creation . If no DHCP server is available, then use static IP configurations. |

⚠️ Warning

In some cases, if you are using the external virtual switch on a Windows Server or client VM, you may need some extra configurations. For more information about nested virtualization configurations, see [Nested virtualization for Azure IoT Edge for Linux on Windows](#).

If you're still having issues with the IP address assignation, try setting up another Windows or Linux virtual machine and assign the same switch and IP configuration. If you have the same issue with the new non-EFLOW VM, the issue is likely with the virtual switch or IP configuration and it's not specific to EFLOW.

Check Domain Name System (DNS) configuration

Your second step when troubleshooting IoT Edge for Linux on Windows networking should be to check the DNS servers assigned to the EFLOW VM. To check the EFLOW VM DNS configuration, see [Networking configuration for Azure IoT Edge for Linux on Windows](#). If address resolution is working, then the issue is likely related to firewall or security configurations on the network.

The EFLOW VM uses the *systemd-resolved* service to manage the DNS resolution. For more information about this service, see [Systemd-resolved](#). To set up a specific DNS server address, you can use the `Set-EflowVmDnsServers` cmdlet. If you need further information about the DNS configuration, you can check the `/etc/systemd/resolved.conf` and the *system-resolved* service using the `sudo systemctl status systemd-resolved` command. Also, you can set a specific DNS server as part of the module configuration, see [Option 2: Set DNS server in IoT Edge deployment per module](#).

The address resolution could fail for multiple reasons. First, the DNS servers could be configured correctly, however they can't be reached from the EFLOW VM. If the DNS servers respond to ICMP ping traffic, you can try pinging the DNS servers to check network connectivity.

1. Start an elevated *PowerShell* session using [Run as Administrator](#).
2. Connect to the EFLOW virtual machine.

```
PowerShell
```

```
Connect-EflowVm
```

3. Ping the DNS server address and check the response.

```
Bash
```

```
ping <DNS-Server-IP-Address>
```

💡 Tip

If the server is reachable, you should get a response, and your issue may be related to other DNS server configurations. If there's no response, then you probably have a connection issue to the server.

Second, some network environments will limit the access of the DNS servers to specific allowlist addresses. If so, first make sure that you can access the DNS server from the Windows host OS, and then check with your networking team if you need to add the EFLOW IP address to an allowlist.

Finally, some network environments block public DNS servers, like Google DNS (8.8.8.8 and 8.8.4.4). If so, talk with your network environment team to define a valid DNS server, and then set it up using the `Set-EflowVmDnsServers` cmdlet.

Check your firewall and port configuration rules

Azure IoT Edge for Linux on Windows allows communication from an on-premises server to Azure cloud using supported Azure IoT Hub protocols. For more information about IoT Hub protocols, see [choosing a communication protocol](#). For more information about IoT Hub firewall and port configurations, see [Troubleshoot your IoT Edge device](#).

The IoT Edge for Linux on Windows is still dependent on the underlying Windows host OS and the network configuration. Hence, it's imperative to ensure proper network and firewall rules are set up for secure edge to cloud communication. The following table can be used as a guideline when configuration firewall rules for the underlying servers where Azure IoT Edge for Linux on Windows runtime is hosted:

[] Expand table

| Protocol | Port | Incoming | Outgoing | Guidance |
|----------|------|----------------------|----------------------|--|
| MQTT | 8883 | BLOCKED
(Default) | BLOCKED
(Default) | Configure <i>Outgoing (Outbound)</i> to be <i>Open</i> when using MQTT as the communication protocol.

1883 for MQTT isn't supported by IoT Edge. - Incoming (Inbound) connections should be blocked. |
| AMQP | 5671 | BLOCKED
(Default) | OPEN
(Default) | Default communication protocol for IoT Edge.

Must be configured to be <i>Open</i> if Azure IoT Edge isn't configured for other supported protocols or AMQP is the desired communication protocol.

5672 for AMQP isn't supported by IoT Edge.

Block this port when Azure IoT Edge uses a different IoT Hub supported protocol. |
| HTTPS | 443 | BLOCKED
(Default) | OPEN
(Default) | Incoming (Inbound) connections should be blocked.

Configure <i>Outgoing (Outbound)</i> to be <i>Open</i> on port 443 for IoT Edge provisioning. This configuration is |

| Protocol | Port | Incoming | Outgoing | Guidance |
|----------|------|----------|----------|---|
| | | | | <p>required when using manual scripts or Azure IoT Device Provisioning Service (DPS).</p> <p><i>Incoming (Inbound) connection</i> should be <i>Open</i> only for two specific scenarios:</p> <ol style="list-style-type: none"> 1. If you have a transparent gateway with downstream devices that may send method requests. In this case, port 443 doesn't need to be open to external networks to connect to IoT Hub or provide IoT Hub services through Azure IoT Edge. Thus the incoming rule could be restricted to only open <i>Incoming (Inbound)</i> from the internal network. 2. For <i>client to device (C2D)</i> scenarios. <p>80 for HTTP isn't supported by IoT Edge.</p> <p>If non-HTTP protocols (for example, AMQP or MQTT) can't be configured in the enterprise; the messages can be sent over WebSockets. Port 443 is used for WebSocket communication in that case.</p> |

ⓘ Note

If you are using an external virtual switch, make sure to add the appropriate firewall rules for the module port mappings you're using inside the EFLOW virtual machine.

For more information about EFLOW VM firewall, see [IoT Edge for Linux on Windows Security](#). To check the EFLOW virtual machine rules, use the following steps:

1. Start an elevated *PowerShell* session using [Run as Administrator](#).
2. Connect to the EFLOW virtual machine.

```
PowerShell
```

```
Connect-EflowVm
```

3. List the [iptables](#) firewall rules.

```
PowerShell
```

```
sudo iptables -L
```

To add a firewall rule to the EFLOW VM, you can use the [EFLOW Util - Firewall Rules](#) sample PowerShell cmdlets. Also, you can achieve the same rules creation by following

these steps:

1. Start an elevated *PowerShell* session using **Run as Administrator**.
2. Connect to the EFLOW virtual machine

```
PowerShell
```

```
Connect-EflowVm
```

3. Add a firewall rule to accept incoming traffic to *<port>* of *<protocol>* (*udp* or *tcp*) traffic.

```
PowerShell
```

```
sudo iptables -A INPUT -p <protocol> --dport <port> -j ACCEPT
```

4. Finally, persist the rules so that they're recreated on every VM boot

```
PowerShell
```

```
sudo iptables-save | sudo tee /etc/systemd/scripts/ip4save
```

Check other configurations

There are multiple other reasons why network communication could fail. The following section will just list a couple of issues that users have encountered in the past.

- **EFLOW virtual machine won't respond to ping (ICMP traffic) requests.**

By default ICMP ping traffic response is disabled on the EFLOW VM firewall. To respond to ping requests, allow the ICMP traffic by using the following PowerShell cmdlet:

```
Invoke-EflowVmCommand "sudo iptables -A INPUT -p icmp --icmp-type 8 -s 0/0 -m state --state NEW,ESTABLISHED,RELATED -j ACCEPT"
```

- **Failed device discovery using multicast traffic.**

First, to allow device discovery via multicast, the Hyper-V VM must be configured to use an external switch. Second, the IoT Edge custom module must be configured to use the host network via the container create options:

```
JSON
```

```
{  
  "HostConfig": {  
    "NetworkMode": "host"  
  },  
  "NetworkingConfig": {  
    "EndpointsConfig": {  
      "host": {}  
    }  
  }  
}
```

- **Firewall rules added, but traffic still not able to reach the IoT Edge module.**

If communication is still not working after adding the appropriate firewall rules, try completely disabling the firewall to troubleshoot.

Bash

```
sudo iptables -F  
sudo iptables -X  
sudo iptables -P INPUT ACCEPT  
sudo iptables -P OUTPUT ACCEPT  
sudo iptables -P FORWARD ACCEPT
```

Once finished, reboot the VM ([Stop-EflowVm](#) and [Start-EflowVm](#)) to get the EFLOW VM firewall back to normal state.

- **Can't connect to internet when using multiple NICs.**

Generally, this issue is related to a routing problem. Check [How to configure Azure IoT Edge for Linux on Windows Industrial IoT & DMZ configuration](#) to set up a static route.

Next steps

Do you think that you found a bug in the IoT Edge platform? [Submit an issue](#) so that we can continue to improve.

If you have more questions, create a [Support request](#) for help.

Feedback

Was this page helpful?

 Yes

 No

Provide product feedback ↗

Common issues and resolutions for Azure IoT Edge for Linux on Windows

Article • 06/06/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Use this article to help resolve common issues that can occur when deploying IoT Edge for Linux on Windows solutions.

Installation and Deployment

The following section addresses the common errors when installing the EFLOW MSI and deploying the EFLOW virtual machine. Ensure you have an understanding of the following EFLOW concepts:

- [Azure IoT Edge for Linux on Windows prerequisites](#)
- [Nested virtualization for Azure IoT Edge for Linux on Windows](#)
- [Networking configuration for Azure IoT Edge for Linux on Windows](#)
- [Azure IoT Edge for Linux on Windows virtual switch creation](#)
- [PowerShell functions for IoT Edge for Linux on Windows](#)

 Expand table

| Error | Error Description | Solution |
|--|--|--|
| HNS API version X doesn't meet minimum version | EFLOW uses HCS/HNS to create the virtual machine on client SKUs. The minimum HNS version is 9.2. | If you're using a Windows version 20H1 or later, the HCS/HNS API should meet the requirement. If you're using Windows Client RS5 (17763), verify you have the latest Windows update. |

| Error | Error Description | Solution |
|---|---|---|
| Can't find WSSDAGENT service!
WssdAgent is unreachable, update has failed. | The <i>WSSDAgent</i> is the EFLOW component that creates and manages the lifecycle of the VM. <i>WSSDAgent</i> runs a service on the Windows host OS. If the service isn't running, the VM communication and lifecycle fails. | Verify the <i>WSSDAgent</i> service is running on the Windows host OS by opening an elevated PowerShell session and running the command <code>Get-Service -Name wssdagent</code> . If <i>WSSDAgent</i> isn't running, try starting the service manually using the cmdlet: <code>Start-Service -name WSSDAgent</code> . If it doesn't start, share the content under <code>C:\ProgramData\wssdagent</code> . |
| Expected image '\$vhdPathBackup' is missing | When installing EFLOW current release (CR), the user can provide the data partition size using the <i>vmDataSize</i> . If specified, EFLOW resizes the VHDX. The error occurs if the VHDX file isn't found during resizing. | Verify the VHDX file wasn't deleted or moved from the original location. |
| Installation of Hyper-V, Hyper-V Management PowerShell or OpenSSH failed. Please install manually and restart deployment. Aborting... | EFLOW installation has required prerequisites to deploy | Ensure that all required prerequisites are met.
<i>PowerShell</i> : Close PowerShell session and open a new one.
If you have multiple |

| Error | Error Description | Solution |
|--|--|--|
| | <p>the virtual machine. If one the prerequisites aren't met, the <code>Deploy-Eflow</code> cmdlet fails.</p> | <p>installations, make sure to have the correct module imported. Try a different version of PowerShell.</p> <p><i>Hyper-V:</i> For more information EFLOW Hyper-V support, see Azure IoT Edge for Linux on Windows Nested Virtualization.</p> <p><i>OpenSSH:</i> If you're using your own custom installation, check <code>customSsh</code> parameter during deployment.</p> |
| <p>\$feature isn't available in this Windows edition. \$featureversion could not be enabled. Please add '\$featureversion' optional feature in settings and restart the deployment.</p> | <p>When deploying EFLOW, if one of the prerequisites it's not met, the installer tries to install it. If this feature isn't available or the feature installation fails, the EFLOW deployment fails.</p> | <p>Ensure that all required prerequisites are met.</p> <p><i>PowerShell:</i> Close PowerShell session and open a new one. If you have multiple installations, make sure to have the correct module imported. Try a different version of PowerShell.</p> <p><i>Hyper-V:</i> For more information EFLOW Hyper-V support, see Azure IoT Edge for Linux on Windows Nested Virtualization.</p> <p><i>OpenSSH:</i> If you're using your own custom installation, check <code>customSsh</code> parameter during deployment.</p> |
| <p>Hyper-V properties indicate the Hyper-V component is not functional (property <code>HyperVRequirementVirtualizationFirmwareEnabled</code> is false)</p> <p>Hyper-V properties indicate the Hyper-V component is not functional (property <code>HyperVisorPresent</code> is false).</p> <p>Hyper-V core services not running (<code>vmms</code>, <code>vmcompute</code>, <code>hvhost</code>). Ensure Hyper-V is configured properly and capable of starting virtual machines.</p> | <p>These errors are related to Hyper-V virtualization technology stack and services. The EFLOW virtual machine requires several</p> | <p>For more information EFLOW Hyper-V support, see Azure IoT Edge for Linux on Windows Nested Virtualization.</p> |

| Error | Error
Description | Solution |
|---|--|---|
| | <p>Hyper-V services in order to create and run the virtual machine. If one of these services isn't available, the installation fails.</p> | |
| wssdagent unreachable, please retry... | <p>The <i>WSSDAgent</i> is the EFLOW component that creates and manages the lifecycle of the VM. <i>WSSDAgent</i> runs a service on the Windows host OS.</p> | <p>Verify the <i>WSSDAgent</i> service is running on the Windows host OS by opening an elevated PowerShell session and running the command <code>Get-Service -Name wssdagent</code>. If <i>WSSDAgent</i> isn't running, try starting the service manually using the cmdlet: <code>Start-Service -name WSSDAgent</code>. If it doesn't start, share the content under <code>C:\ProgramData\wssdagent</code>.</p> |
| Virtual machine configuration could not be retrieved from wssdagent | <p>Find the EFLOW configuration yaml files under the EFLOW root installation folder. For example, if the default installation directory was used, the configuration files should be in the <code>C:\Program</code></p> | <p>Check if the directory was deleted or moved. If the directory isn't available, the VM can't be created. EFLOW reinstallation is needed.</p> |

| Error | Error Description | Solution |
|---|---|---|
| | <i>Files\Azure IoT Edge\yaml</i> directory. | |
| <p>An existing virtual machine was detected. To redeploy the virtual machine, please uninstall and re-install \$eflowProductName.</p> <p>Virtual machine '\$name' already exists. You must remove '\$name' virtual machine first.</p> | <p>During EFLOW deployment, the installer checks if there's an EFLOW VM created from previous installations. In some cases, if the installation fails in its final steps, the VM was created and it's still running in the Windows host OS.</p> | <p>Make sure to completely uninstall EFLOW before starting a new installation. If you want to remove the Azure IoT Edge for Linux on Windows installation from your device, use the following commands.</p> <ol style="list-style-type: none"> 1. In Windows, open Settings 2. Select Add or Remove Programs 3. Select Azure IoT Edge LTS app 4. Select Uninstall |
| <p>Creating storage vhd (file: \$(\$config["imageNameEflowVm"])) failed</p> | <p>Error when creating or resizing the EFLOW virtual machine</p> | <p>Check EFLOW installation logs <i>C:\Program Files\Azure IoT Edge</i> and <i>WSSDAgent</i> logs <i>C:\ProgramData\wssdagent</i> for more information.</p> |
| <p>Error: Virtual machine creation failed!</p> <p>Failed to retrieve virtual machine name.</p> | <p>Error related to virtual machine creation by <i>WSSDAgent</i>. Installer will try removing the VM and mark the installation as failed.</p> | <p>Verify the <i>WSSDAgent</i> service is running on the Windows host OS by opening an elevated PowerShell session and running the command <code>Get-Service -Name wssdagent</code>. If <i>WSSDAgent</i> isn't running, try starting the service manually using the cmdlet: <code>Start-Service -name WSSDAgent</code>. If it doesn't start,</p> |

| Error | Error Description | Solution |
|---|--|--|
| | | share the content under <code>C:\ProgramData\wssdagent</code> . |
| This Windows device does not meet the minimum requirements for Azure EFLOW. Please refer to https://aka.ms/AzEFLOW-Requirements for system requirements | During EFLOW deployment, the <code>Deploy-EFlow</code> PowerShell cmdlet checks that all the prerequisites are met. Specifically, Windows SKUs are checked (Windows Server 2019, 2022, Windows Client Professional, or Client Enterprise) and the Windows version is at least 17763. | Verify you're using a supported Windows SKU and version. If using Windows version 17763, ensure to have all the updates applied. |
| Not enough memory available. | There isn't enough RAM memory to create the EFLOW VM with the allocated VM memory. By default, the virtual machine has 1024 MB assigned. The Windows | Check the Windows host OS memory available and use the <code>memoryInMb</code> parameter during <code>Deploy-Eflow</code> for custom memory assignment. For more information about <code>Deploy-EFlow</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |

| Error | Error
Description | Solution |
|---|---|--|
| | <p>host OS needs to have at least X MB free to assign that memory to the VM.</p> | |
| <p>Not enough CPU cores available.</p> | <p>There isn't enough CPU cores available to create the EFLOW VM with the allocated cores. By default, the virtual machine will have one core assigned.</p> <p>The Windows host OS needs to have at least one core to assign that core to the EFLOW VM.</p> | <p>Check the Windows host OS CPU cores available and use the <i>cpuCore</i> parameter during <code>Deploy-Eflow</code> for custom memory assignment. For more information about <code>Deploy-Eflow</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows.</p> |
| <p>Not enough disk space is available on drive '\$drive'.</p> | <p>There isn't enough storage available to create the EFLOW VM with the allocated storage size. By default, the VM will use ~18 GB of storage.</p> <p>The</p> | <p>Check the Windows host free storage available and use the <i>vmDiskSize</i> parameter during <code>Deploy-Eflow</code> for custom storage size. For more information about <code>Deploy-Eflow</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows.</p> |

| Error | Error
Description | Solution |
|--|--|--|
| | Windows host OS needs to have at least 18 GB of free storage to assign that storage to the EFLOW VM VHDX. | |
| <p>Signature is not valid (file: \$filePath, signature status: \${signature.Status})</p> <p>Signature is missing (file: \$filePath)! could not track signature to the microsoft root certificate</p> | <p>The signature of the file can't be found or it's invalid. EFLOW PSM and EFLOW updates are all signed using Microsoft certificates. If the Microsoft code or Microsoft CA certificates are not available in the Windows host OS, the validation fails.</p> | <p>Verify all contents were downloaded from official Microsoft sites. Also, if the necessary certificates aren't part of the Windows host, check Install EFLOW necessary certificates.</p> |

Provisioning and IoT Edge runtime

The following section addresses the common errors when provisioning the EFLOW virtual machine and interact with the IoT Edge runtime. Ensure you have an understanding of the following EFLOW concepts:

- [What is Azure IoT Hub Device Provisioning Service?](#)
- [Understand the Azure IoT Edge runtime and its architecture](#)
- [Troubleshoot your IoT Edge device](#)

| Error | Error Description | Solution |
|--|---|---|
| Action aborted by user | For some of the EFLOW PowerShell cmdlets, there's user interaction and confirmation needed. | - |
| Error, device connection string not provided.
Only the device connection string for <i>ManualConnectionString</i> provisioning may be specified. | Incorrect parameters used when using <i>ManualConnectionString</i> device provisioning. | For more information about the <code>Provision-EflowVm</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |
| IoT Hub Hostname, Device ID, and/or identity cert/pk parameters for ManualX509 provisioning not specified
Device connection string, scope ID, registration ID, and symmetric key may not be specified for DpsX509 provisioning
Certificate and private key file for ManualX509 provisioning not found (expected at \$identityCertPath and \$identityPrivKeyPath) | Incorrect parameters used when using <i>ManualX509</i> device provisioning. | For more information about the <code>Provision-EflowVm</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |
| Scope ID for DpsTpm provisioning not specified
Only scope ID and registration ID (optional) for DpsTpm provisioning may be specified | Incorrect parameters used when using <i>DpsTpm</i> device provisioning. | For more information about the <code>Provision-EflowVm</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |
| Scope ID, registration ID or symmetric key missing for DpsSymmetricKey provisioning
globalEndpoint not specified
Only scope ID, registration ID or symmetric key for DpsSymmetricKey provisioning may be specified | Incorrect parameters used when using <i>DpsSymmetricKey</i> device provisioning. | For more information about the <code>Provision-EflowVm</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |
| Virtual machine does not exist, deploy it first | The EFLOW MSI was installed, however the EFLOW virtual machine was never deployed. | Deploy the EFLOW VM using the <code>Deploy-Eflow</code> PowerShell cmdlet. |

| Error | Error Description | Solution |
|--|--|---|
| Aborting, iotedge was previously provisioned (headless mode)! | The EFLOW VM was previously provisioned and <i>headless</i> mode isn't supported when reprovisioning. | The issue is fixed and now - <i>headless</i> mode is supported with reprovisioning |
| Provisioning aborted by user | The EFLOW VM was previously provisioned and the user needs to confirm they want to reprovision. | User must accept reprovisioning to continue with the provisioning process. |
| Failed to provision
Failed to provision config.toml.
Please provision manually.
iotedge service not running after provisioning, please investigate manually | The EFLOW provisioning information was correctly configured inside the EFLOW VM, but the IoT Edge daemon was not able to provision the device with the cloud provisioning service. | Check the Azure IoT Edge runtime logs. First, connect to the EFLOW virtual machine using the Connect-EflowVm PowerShell cmdlet then follow Troubleshoot your IoT Edge device to retrieve the IoT Edge logs. |
| Unexpected return from 'sudo iotedge list'
Retrieving iotedge check output from: "\$vmName" | The execution of <code>sudo iotedge list</code> command inside the EFLOW VM returned an unexpected payload. Generally this is related to IoT Edge service not running correctly inside the EFLOW VM. | Check the Azure IoT Edge runtime logs. First, connect to the EFLOW virtual machine using the Connect-EflowVm PowerShell cmdlet then follow Troubleshoot your IoT Edge device to get the IoT Edge logs. |
| TPM 2.0 is required to enable DpsTpm | TPM passthrough only works with TPM 2.0 compatible hardware. This could be caused by a physical TPM or if the EFLOW VM is running using nested virtualization using vTPM on the Windows host OS. | Make sure the Windows host OS has a valid TPM 2.0, check Enable TPM 2.0 on your PC . |
| TPM provisioning information not available! | The TPM passthrough binary inside the EFLOW VM could not get the TPM information from the Windows host OS. This error is probably related to a communication error with the EFLOWProxy. | Ensure that the <i>EFLOW Proxy Service</i> service is running using the PowerShell cmdlet <code>Get-Service -name "EFLOW Proxy Service"</code> . If not running, check the Event logs. Open the Event Viewer > Application and service logs -> Azure IoT Edge for Linux on Windows . |

Interaction with the VM

The following section addresses the common errors when interacting with the EFLOW virtual machine, and configure the EFLOW device passthrough options. Ensure you have an understanding of the following EFLOW concepts:

- [PowerShell functions for IoT Edge for Linux on Windows](#)
- [GPU acceleration for Azure IoT Edge for Linux on Windows](#)

 [Expand table](#)

| Error | Error Description | Solution |
|--|--|--|
| Can't process request, EFLOW VM is OFF! | When trying to apply a configuration to the EFLOW virtual machine, the VM must be turned on. If the EFLOW VM is off, then the SSH channel will fail, and no communication is possible with the VM. | Start the EFLOW VM using the <code>Start-EflowVm</code> PowerShell cmdlet. For more information about the <code>Start-EflowVm</code> cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |
| Virtual machine name could not be retrieved from wssdagent
Error: More than one virtual machine found | The WSSDAgent service couldn't find the EFLOW virtual machine. | Verify the EFLOW VM is started and running, use the PowerShell cmdlet <code>Start-EflowVm</code> . If using a client SKU, use the <code>hcsdiag list</code> cmdlet and find the line that has <code>wssdagent</code> after the GUID of the VM then check the state. If using a server SKU, go to Hyper-V manager and verify there's a VM with the name <code>Windows-hostname-EFLOW</code> then check the state. |
| Unable to connect virtual machine with SSH. Aborting.. | The Windows host OS could not establish an SSH connection with the EFLOW VM to execute the necessary commands or copy files. Generally, this | Try the PowerShell cmdlet <code>Get-EflowVmAddr</code> and check if the <code>IP4Address</code> assigned to the VM is correct. For more information about networking configurations, see Azure IoT Edge for Linux on Windows networking . |

| Error | Error Description | Solution |
|--|---|--|
| | issue is related to a networking problem between Windows and the virtual machine. | |
| Unexpected return stats from virtual machine | The execution of <code>Get-EflowVm</code> PowerShell cmdlet checks the status of the virtual machine. If the communication with the virtual machine fails, or some of the Linux bash commands inside the VM fail, the cmdlet fails. | Check the EFLOW VM connection using <code>Connect-EflowVm</code> PowerShell cmdlet and try manually running the VM stats bash commands inside the VM. |
| TPM provisioning was not enabled! | To get the TPM provisioning information for the TPM, the EFLOW TPM passthrough must be enabled. If TPM passthrough isn't enabled, the cmdlet fails. | Enable TPM passthrough before getting the TPM information. Use the <code>Set-EflowVmFeature</code> PowerShell cmdlet to enable the TPM passthrough. For more information about <code>Set-EflowVmFeature</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |
| Unknown feature '\$feature' is provided. | The <code>Set-EflowVmFeature</code> cmdlet supports <code>DpsTpm</code> and <code>Defender</code> as the two features that can be enabled or disabled. | For more information about <code>Set-EflowVmFeature</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |
| Unsupported DDA Type: \$gpuName | Currently, GPU DDA is only supported for <i>NVIDIA Tesla T4</i> and <i>NVIDIA A2</i> . If the user provides another GPU name, the GPU passthrough fails. | Verify all the GPU prerequisites are met. For more information about EFLOW GPU support, check Azure IoT Edge for Linux on Windows GPU passthrough . |

| Error | Error Description | Solution |
|--|---|--|
| Invalid GPU configuration requested,
Passthrough enabled but requested
gpuCount == \$gpuCount
GPU PassthroughType
'\$gpuPassthroughType' not supported by
'\$script:WssdProviderVmms' WssdProvider
Requested GPU configuration cannot be
supported by Host,
GPU '\$gpuName' not available
Requested GPU configuration cannot be
supported by Host,
not enough GPUs available -
Requested(\$gpuCount),
Available(\$selectedGpuDevice.Count))
Requested GPU configuration cannot be
supported by Host,
GPU PassthroughType
'\$gpuPassthroughType' not supported
Invalid GPU configuration requested,
Passthrough disabled but gpuCount > 0" | These errors are generally related to one or more the GPU dependencies not being met. | Make sure all the GPU prerequisites are met. For more information about EFLOW GPU support, check Azure IoT Edge for Linux on Windows GPU passthrough . |

Networking

The following section addresses the common errors related to EFLOW networking and communication between the EFLOW virtual machine and the Windows host OS. Ensure you have an understanding of the following EFLOW concepts:

- [Azure IoT Edge for Linux on Windows networking](#)
- [Networking configuration for Azure IoT Edge for Linux on Windows](#)
- [Azure IoT Edge for Linux on Windows virtual switch creation](#)

[\[+\] Expand table](#)

| Error | Error Description | Solution |
|---|---|--|
| Installation of virtual switch failed
The virtual switch '\$switchName' of type '\$switchType' was not found | When creating the EFLOW VM, there's a check that the virtual switch provided exists and has the correct type. If using no parameter, the installation uses the default switch provided by the Windows client. | Check that the virtual switch being used is part of the Windows host OS. You can check the virtual switches using the PowerShell cmdlet <code>Get-VmSwitch</code> . For more information about networking configurations, see Azure IoT Edge for Linux on Windows networking . |

| Error | Error Description | Solution |
|--|--|---|
| The virtual switch '\$switchName' of type '\$switchType' is not supported on current host OS | When using Windows client SKUs, external/default switches are supported. However, when using Windows Server SKUs, external/internal switches are supported. | For more information about networking configurations, see Azure IoT Edge for Linux on Windows networking . |
| Cannot set Static IP on ICS type virtual switch (Default Switch) | The <i>default switch</i> is a virtual switch that's provided in the Windows client SKUs after installing Hyper-V. This switch already has a DHCP server for <i>IP4Address</i> assignation and for security reasons doesn't support a static IP. | If using the <i>default switch</i> , you can either use the <code>Get-EflowVmAddr</code> cmdlet or use the hostname of the EFLOW VM to get the VM <i>IP4Address</i> . If using the hostname, try using <code>Windows-hostname-EFLOW.mshome.net</code> . For more information about networking configurations, see Azure IoT Edge for Linux on Windows networking . |
| \$dnsServer is not a valid IP4 address | The <i>Set-EflowVmDnsServers</i> cmdlet expects a list of valid <i>IP4Addresses</i> | Verify you provided a valid list of addresses. You can check the Windows host OS DNS servers by using the PowerShell cmdlet <code>ipconfig /all</code> and then looking for the entry <i>DNS Servers</i> . For example, if you wanted to set two DNS servers with IPs 10.0.1.2 and 10.0.1.3, use the <code>Set-EflowVmDnsServers -dnsServers @("10.0.1.2", "10.0.1.3")</code> cmdlet. |
| Could not retrieve IP address for virtual machine
Virtual machine name could not be retrieved, failed to get IP/MAC address
Failed to acquire MAC address for virtual machine
Failed to acquire IP address for virtual machine
Unable to obtain host routing.
Connectivity test to \$computerName failed. | Caused by connectivity issues with the EFLOW virtual machine. The errors are generally related to an IP address change (if using static IP) or failure to assign an IP if using DHCP server. | Make sure to use the appropriate networking configuration. If there's a valid DHCP server, you can use DHCP assignation. If using static IP, make sure the IP configuration is correct (all three parameters: <i>ip4Address</i> , <i>ip4GatewayAddress</i> and <i>ip4PrefixLength</i>) and the address isn't being used by another device in the network. For more information about networking configurations, see Azure IoT Edge for Linux on Windows networking . |

| Error | Error Description | Solution |
|---|---|---|
| wssdagent does not have the expected vnet resource provisioned.
Missing EFLOW-VM guest interface for (\$vnicName) | | |
| No adapters associated with the switch '\$vnetName' are found.
No adapters associated with the device ID '\$adapterGuid' are found
No adapters associated with the adapter name '\$name' are found.
Network '\$vswitchName' does not exist | Caused by a network communication error between the Windows host OS and the EFLOW virtual machine. | Ensure you can reach the EFLOW VM and establish an SSH channel. Use the <code>Connect-EflowVm</code> PowerShell cmdlet to connect to the virtual machine. If connectivity fails, reboot the EFLOW VM and check again. |
| ip4Address & ip4PrefixLength are required for StaticIP! | During EFLOW VM deployment or when adding multiple NICs, if using static IP, the three static ip parameters are needed: <i>ip4Address</i> , <i>ip4GatewayAddress</i> , <i>ip4PrefixLength</i> . | For more information about <code>Deploy-EFlow</code> PowerShell cmdlet, see PowerShell functions for IoT Edge for Linux on Windows . |
| Found multiple VMMS switches with name '\$switchName' of type '\$switchType' | There are two or more virtual switches with the same name and type. This environment conflicts with the EFLOW VM installation and lifecycle of the VM. | Use <code>Get-VmSwitch</code> PowerShell cmdlet to check the virtual switches available in the Windows host and make sure that each {name,type} is unique. |

Next steps

Do you think that you found a bug in the IoT Edge for Linux on Windows? [Submit an issue](#) so that we can continue to improve.

If you have more questions, create a [Support request](#) for help.

PowerShell functions for IoT Edge for Linux on Windows

Article • 06/10/2024

Applies to:  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

Understand the PowerShell functions that deploy, provision, and get the status of your IoT Edge for Linux on Windows (EFLOW) virtual machine.

Prerequisites

The commands described in this article are from the `AzureEFLOW.psm1` file, which can be found on your system in your `WindowsPowerShell` directory under `C:\Program Files\WindowsPowerShell\Modules\AzureEFLOW`.

If you don't have the `AzureEfLow` folder in your PowerShell directory, use the following steps to download and install Azure IoT Edge for Linux on Windows:

1. In an elevated PowerShell session, run each of the following commands to download IoT Edge for Linux on Windows.

- X64/AMD64

PowerShell

```
$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))  
$ProgressPreference = 'SilentlyContinue'  
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_X64" -OutFile  
$msiPath
```

- ARM64

PowerShell

```
$msiPath = $([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))  
$ProgressPreference = 'SilentlyContinue'
```

```
Invoke-WebRequest "https://aka.ms/AzEFLOWMSI_1_4_LTS_ARM64" -OutFile  
$msiPath
```

2. Install IoT Edge for Linux on Windows on your device.

PowerShell

```
Start-Process -Wait msiexec -ArgumentList  
"/i","$([io.Path]::Combine($env:TEMP, 'AzureIoTEdge.msi'))","/qn"
```

You can specify custom installation and VHDX directories by adding `INSTALLDIR="<FULLY_QUALIFIED_PATH>"` and `VHDXDIR="<FULLY_QUALIFIED_PATH>"` parameters to the install command.

3. Set the execution policy on the target device to at least `AllSigned`.

PowerShell

```
Set-ExecutionPolicy -ExecutionPolicy AllSigned -Force
```

Add-EflowNetwork

The **Add-EflowNetwork** command adds a new network to the EFLOW virtual machine. This command takes two parameters.

[Expand table](#)

| Parameter | Accepted values | Comments |
|-------------|----------------------------|--|
| vswitchName | Name of the virtual switch | Name of the virtual switch assigned to the EFLOW VM. |
| vswitchType | Internal or External | Type of the virtual switch assigned to the EFLOW VM. |

It returns an object that contains four properties:

- Name
- AllocationMethod
- Cidr
- Type

For more information, use the command `Get-Help Add-EflowNetwork -full`.

Add-EflowVmEndpoint

The **Add-EflowVmEndpoint** command adds a new network endpoint to the EFLOW virtual machine. Use the optional parameters to set a Static IP.

[+] Expand table

| Parameter | Accepted values | Comments |
|-------------------|--|--|
| vswitchName | Name of the virtual switch | Name of the virtual switch assigned to the EFLOW VM. |
| vendpointName | Name of the virtual endpoint | Name of the virtual endpoint assigned to the EFLOW VM. |
| ip4Address | IPv4 Address in the range of the DHCP Server Scope | Static Ipv4 address of the EFLOW VM. |
| ip4PrefixLength | IPv4 Prefix Length of the subnet | Ipv4 subnet prefix length, only valid when static Ipv4 address is specified. |
| ip4GatewayAddress | IPv4 Address of the subnet gateway | Gateway Ipv4 address, only valid when static Ipv4 address is specified. |

It returns an object that contains four properties:

- Name
- MacAddress
- HealthStatus
- IpConfiguration

For more information, use the command `Get-Help Add-EflowVmEndpoint -full`.

Add-EflowVmSharedFolder

The **Add-EflowVmSharedFolder** command allows sharing one or more Windows host OS folders with the EFLOW virtual machine.

[+] Expand table

| Parameter | Accepted values | Comments |
|-----------------------|-----------------|---|
| sharedFoldersJsonPath | String | Path to the Shared Folders JSON configuration file. |

The JSON configuration file must have the following structure:

- **sharedFolderRoot** : Path to the Windows root folder that contains all the folders to be shared with the EFLOW virtual machine.
- **hostFolderPath**: Relative path (to the parent root folder) of the folder to be shared with the EFLOW VM.
- **readOnly**: Defines if the shared folder is writeable or read-only from the EFLOW virtual machine - Values: **false** or **true**.
- **targetFolderOnGuest** : Folder path inside the EFLOW virtual machine where Windows host OS folder is mounted.

JSON

```
[
  {
    "sharedFolderRoot": "<shared-folder-root-windows-path>",
    "sharedFolders": [
      {
        "hostFolderPath": "<path-shared-folder>",
        "readOnly": "<read-only>",
        "targetFolderOnGuest": "<linux-mounting-point>"
      }
    ]
  }
]
```

For more information, use the command `Get-Help Add-EflowVmSharedFolder -full`.

Connect-EflowVm

The **Connect-EflowVm** command connects to the virtual machine using SSH. The only account allowed to SSH to the virtual machine is the user that created it.

This command only works on a PowerShell session running on the host device. It won't work when using Windows Admin Center or PowerShell ISE.

For more information, use the command `Get-Help Connect-EflowVm -full`.

Copy-EflowVmFile

The **Copy-EflowVmFile** command copies file to or from the virtual machine using SCP. Use the optional parameters to specify the source and destination file paths and the direction of the copy.

The user **iotedge-user** must have read permission to any origin directories or write permission to any destination directories on the virtual machine.

[Expand table](#)

| Parameter | Accepted values | Comments |
|-----------|----------------------------------|---|
| fromFile | String representing path to file | Defines the file to be read from. |
| toFile | String representing path to file | Defines the file to be written to. |
| pushFile | None | This flag indicates copy direction. If present, the command pushes the file to the virtual machine. If absent, the command pulls the file from the virtual machine. |

For more information, use the command `Get-Help Copy-EflowVMFile -full`.

Deploy-Eflow

The **Deploy-Eflow** command is the main deployment method. The deployment command creates the virtual machine, provisions files, and deploys the IoT Edge agent module. While none of the parameters are required, they can be used to modify settings for the virtual machine during creation.

[Expand table](#)

| Parameter | Accepted values | Comments |
|-------------------------|--|---|
| acceptEula | Yes or No | A shortcut to accept/deny EULA and bypass the EULA prompt. |
| acceptOptionalTelemetry | Yes or No | A shortcut to accept/deny optional telemetry and bypass the telemetry prompt. |
| cpuCount | Integer value between 1 and the device's CPU cores | Number of CPU cores for the VM.

Default value: 1 vCore. |
| memoryInMB | Integer even value between 1024 and the maximum amount of free memory of the device | Memory allocated for the VM.

Default value: 1024 MB. |
| vmDiskSize | Between 21 GB and 2 TB | Maximum logical disk size of the dynamically expanding virtual hard disk. |

| Parameter | Accepted values | Comments |
|--------------------|--|--|
| | | Default value: 29 GB. |
| | | Note: Either <i>vmDiskSize</i> or <i>vmDataSize</i> can be used, but not both together. |
| vmDataSize | Between 2 GB and 2 TB | Maximum data partition size of the resulting hard disk, in GB. |
| | | Default value: 10 GB. |
| | | Note: Either <i>vmDiskSize</i> or <i>vmDataSize</i> can be used, but not both together. |
| vmLogSize | Small or Large | Specify the log partition size.
Small = 1GB, Large = 6GB. |
| | | Default value: Small. |
| vswitchName | Name of the virtual switch | Name of the virtual switch assigned to the EFLOW VM. |
| vswitchType | Internal or External | Type of the virtual switch assigned to the EFLOW VM. |
| ip4Address | IPv4 Address in the range of the DCHP Server Scope | Static Ipv4 address of the EFLOW VM. |
| ip4PrefixLength | IPv4 Prefix Length of the subnet | Ipv4 subnet prefix length, only valid when static Ipv4 address is specified. |
| ip4GatewayAddress | IPv4 Address of the subnet gateway | Gateway Ipv4 address, only valid when static Ipv4 address is specified. |
| gpuName | GPU Device name | Name of GPU device to be used for passthrough. |
| gpuPassthroughType | DirectDeviceAssignment , ParaVirtualization , or none (CPU only) | GPU Passthrough type |
| gpuCount | Integer value between 1 and the number of the device's GPU cores | Number of GPU devices for the VM. |
| | | Note: If using ParaVirtualization, make sure to set <i>gpuCount</i> = 1 |

| Parameter | Accepted values | Comments |
|-----------------------|-----------------|---|
| customSsh | None | Determines whether user wants to use their custom OpenSSH.Client installation. If present, ssh.exe must be available to the EFLOW PSM |
| sharedFoldersJsonPath | String | Path to the Shared Folders JSON configuration file. |

For more information, use the command `Get-Help Deploy-Eflow -full`.

Get-EflowHostConfiguration

The **Get-EflowHostConfiguration** command returns the host configuration. This command takes no parameters. It returns an object that contains four properties:

- FreePhysicalMemoryInMB
- NumberOfLogicalProcessors
- DiskInfo
- GpuInfo

For more information, use the command `Get-Help Get-EflowHostConfiguration -full`.

Get-EflowLogs

The **Get-EflowLogs** command collects and bundles logs from the IoT Edge for Linux on Windows deployment and installation. It outputs the bundled logs in the form of a `.zip` folder.

For more information, use the command `Get-Help Get-EflowLogs -full`.

Get-EflowNetwork

The **Get-EflowNetwork** command returns a list of the networks assigned to the EFLOW virtual machine. Use the optional parameter to get a specific network.

[\[+\] Expand table](#)

| Parameter | Accepted values | Comments |
|-------------|----------------------------|--|
| vswitchName | Name of the virtual switch | Name of the virtual switch assigned to the EFLOW VM. |

It returns a list of objects that contains four properties:

- Name
- AllocationMethod
- Cidr
- Type

For more information, use the command `Get-Help Get-EflowNetwork -full`.

Get-EflowVm

The **Get-EflowVm** command returns the virtual machine's current configuration. This command takes no parameters. It returns an object that contains four properties:

- VmConfiguration
- VmPowerState
- EdgeRuntimeVersion
- EdgeRuntimeStatus
- SystemStatistics

To view a specific property in a readable list, run the `Get-EflowVM` command with the property expanded. For example:

PowerShell

```
Get-EflowVM | Select -ExpandProperty VmConfiguration | Format-List
```

For more information, use the command `Get-Help Get-EflowVm -full`.

Get-EflowVmAddr

The **Get-EflowVmAddr** command is used to query the virtual machine's current IP and MAC address. This command exists to account for the fact that the IP and MAC address can change over time.

For additional information, use the command `Get-Help Get-EflowVmAddr -full`.

Get-EflowVmEndpoint

The **Get-EflowVmEndpoint** command returns a list of the network endpoints assigned to the EFLOW virtual machine. Use the optional parameter to get a specific network endpoint.

[+] Expand table

| Parameter | Accepted values | Comments |
|-------------|----------------------------|--|
| vswitchName | Name of the virtual switch | Name of the virtual switch assigned to the EFLOW VM. |

It returns a list of objects that contains four properties:

- Name
- MacAddress
- HealthStatus
- IpConfiguration

For more information, use the command `Get-Help Get-EflowVmEndpoint -full`.

Get-EflowVmFeature

The **Get-EflowVmFeature** command returns the status of the enablement of IoT Edge for Linux on Windows features.

[+] Expand table

| Parameter | Accepted values | Comments |
|-----------|-----------------|------------------------|
| feature | DpsTpm | Feature name to query. |

For more information, use the command `Get-Help Get-EflowVmFeature -full`.

Get-EflowVmName

The **Get-EflowVmName** command returns the virtual machine's current hostname. This command exists to account for the fact that the Windows hostname can change over time.

For more information, use the command `Get-Help Get-EflowVmName -full`.

Get-EflowVmSharedFolder

The **Get-EflowVmSharedFolder** command returns the information about one or more Windows host OS folders shared with the EFLOW virtual machine.

[] Expand table

| Parameter | Accepted values | Comments |
|------------------|-----------------|--|
| sharedfolderRoot | String | Path to the Windows host OS shared root folder. |
| hostFolderPath | String or List | Relative path/path(s) (to the root folder) to the Windows host OS shared folder/s. |

It returns a list of objects that contains three properties:

- **hostFolderPath**: Relative path (to the parent root folder) of the folder shared with the EFLOW VM.
- **readOnly**: Defines if the shared folder is writeable or read-only from the EFLOW virtual machine - Values: **false** or **true**.
- **targetFolderOnGuest**: Folder path inside the EFLOW virtual machine where the Windows folder is mounted.

For more information, use the command `Get-Help Get-EflowVmSharedFolder -full`.

Get-EflowVmTelemetryOption

The **Get-EflowVmTelemetryOption** command displays the status of the telemetry (either **Optional** or **Required**) inside the virtual machine.

For more information, use the command `Get-Help Get-EflowVmTelemetryOption -full`.

Get-EflowVmTpmProvisioningInfo

The **Get-EflowVmTpmProvisioningInfo** command returns the TPM provisioning information. This command takes no parameters. It returns an object that contains two properties:

- Endorsement Key
- Registration ID

For more information, use the command `Get-Help Get-EflowVmTpmProvisioningInfo -full`.

Invoke-EflowVmCommand

The **Invoke-EflowVMCommand** command executes a Linux command inside the virtual machine and returns the output. This command only works for Linux commands that return a finite output. It cannot be used for Linux commands that require user interaction or that run indefinitely.

The following optional parameters can be used to specify the command in advance.

[] Expand table

| Parameter | Accepted values | Comments |
|-------------|-----------------|--|
| command | String | Command to be executed in the VM. |
| ignoreError | None | If this flag is present, ignore errors from the command. |

For more information, use the command `Get-Help Invoke-EflowVmCommand -full`.

Provision-EflowVm

The **Provision-EflowVm** command adds the provisioning information for your IoT Edge device to the virtual machine's IoT Edge `config.yaml` file.

[] Expand table

| Parameter | Accepted values | Comments |
|------------------|--|--|
| provisioningType | <code>ManualConnectionString</code> ,
<code>ManualX509</code> , <code>DpsTPM</code> , <code>DpsX509</code> , or
<code>DpsSymmetricKey</code> | Defines the type of provisioning you wish to use for your IoT Edge device. |
| devConnString | The device connection string of an existing IoT Edge device | Device connection string for manually provisioning an IoT Edge device (<code>ManualConnectionString</code>). |
| iotHubHostname | The host name of an existing IoT hub | Azure IoT Hub hostname for provisioning an IoT Edge device (<code>ManualX509</code>). |
| deviceid | The device ID of an existing IoT Edge device | Device ID for provisioning an IoT Edge device (<code>ManualX509</code>). |

| Parameter | Accepted values | Comments |
|---------------------|--|--|
| scopeId | The scope ID for an existing DPS instance. | Scope ID for provisioning an IoT Edge device (DpsTPM , DpsX509 , or DpsSymmetricKey). |
| symmKey | The primary key for an existing DPS enrollment or the primary key of an existing IoT Edge device registered using symmetric keys | Symmetric key for provisioning an IoT Edge device (DpsSymmetricKey). |
| registrationId | The registration ID of an existing IoT Edge device | Registration ID for provisioning an IoT Edge device (DpsSymmetricKey , DpsTPM). |
| identityCertPath | Directory path | Absolute destination path of the identity certificate on your Windows host machine (ManualX509 , DpsX509). |
| identityPrivKeyPath | Directory path | Absolute source path of the identity private key on your Windows host machine (ManualX509 , DpsX509). |
| globalEndpoint | Device Endpoint URL | URL for Global Endpoint to be used for DPS provisioning. |

For more information, use the command `Get-Help Provision-EflowVm -full`.

Remove-EflowNetwork

The **Remove-EflowNetwork** command removes an existing network attached to the EFLOW virtual machine. This command takes one parameter.

[\[+\] Expand table](#)

| Parameter | Accepted values | Comments |
|-------------|----------------------------|--|
| vswitchName | Name of the virtual switch | Name of the virtual switch assigned to the EFLOW VM. |

For more information, use the command `Get-Help Remove-EflowNetwork -full`.

Remove-EflowVmEndpoint

The **Remove-EflowVmEndpoint** command removes an existing network endpoint attached to the EFLOW virtual machine. This command takes one parameter.

[+] Expand table

| Parameter | Accepted values | Comments |
|---------------|------------------------------|--|
| vendpointName | Name of the virtual endpoint | Name of the virtual endpoint assigned to the EFLOW VM. |

For more information, use the command `Get-Help Remove-EflowVmEndpoint -full`.

Remove-EflowVmSharedFolder

The **Remove-EflowVmSharedFolder** command stops sharing the Windows host OS folder to the EFLOW virtual machine. This command takes two parameters.

[+] Expand table

| Parameter | Accepted values | Comments |
|------------------|-----------------|--|
| sharedfolderRoot | String | Path to the Windows host OS shared root folder. |
| hostFolderPath | String or List | Relative path/path(s) (to the root folder) to the Windows host OS shared folder/s. |

For more information, use the command `Get-Help Remove-EflowVmSharedFolder -full`.

Set-EflowVM

The **Set-EflowVM** command updates the virtual machine configuration with the requested properties. Use the optional parameters to define a specific configuration for the virtual machine.

[+] Expand table

| Parameter | Accepted values | Comments |
|------------|---|---------------------------------|
| cpuCount | Integer value between 1 and the device's CPU cores | Number of CPU cores for the VM. |
| memoryInMB | Integer value between 1024 and the maximum amount of free | Memory allocated for the VM. |

| Parameter | Accepted values | Comments |
|--------------------|--|--|
| | memory of the device | |
| gpuName | GPU device name | Name of the GPU device to be used for passthrough. |
| gpuPassthroughType | DirectDeviceAssignment, ParaVirtualization, or none (no passthrough) | GPU Passthrough type |
| gpuCount | Integer value between 1 and the device's GPU cores | Number of GPU devices for the VM
Note: Only valid when using DirectDeviceAssignment |
| headless | None | If this flag is present, it determines whether the user needs to confirm in case a security warning is issued. |

For more information, use the command `Get-Help Set-EflowVM -full`.

Set-EflowVmDNSServers

The **Set-EflowVmDNSServers** command configures the DNS servers for EFLOW virtual machine.

[\[+\] Expand table](#)

| Parameter | Accepted values | Comments |
|---------------|---|--|
| vendpointName | String value of the virtual endpoint name | Use the <i>Get-EflowVmEndpoint</i> to obtain the virtual interfaces assigned to the EFLOW VM. E.g. DESKTOP-CONTOSO-EflowInterface |
| dnsServers | List of DNS server IPAddress to use for name resolution | E.g. @("10.0.10.1") |

For more information, use the command `Get-Help Set-EflowVmDNSServers -full`.

Set-EflowVmFeature

The **Set-EflowVmFeature** command enables or disables the status of IoT Edge for Linux on Windows features.

[Expand table](#)

| Parameter | Accepted values | Comments |
|-----------|------------------|---|
| feature | DpsTpm, Defender | Feature name to toggle. |
| enable | None | If this flag is present, the command enables the feature. |

For more information, use the command `Get-Help Set-EflowVmFeature -full`.

Set-EflowVmTelemetryOption

The **Set-EflowVmTelemetryOption** command enables or disables the optional telemetry inside the virtual machine.

[Expand table](#)

| Parameter | Accepted values | Comments |
|-------------------|-----------------|---|
| optionalTelemetry | True or False | Whether optional telemetry is selected. |

For more information, use the command `Get-Help Set-EflowVmTelemetryOption -full`.

Start-EflowVm

The **Start-EflowVm** command starts the virtual machine. If the virtual machine is already started, no action is taken.

For more information, use the command `Get-Help Start-EflowVm -full`.

Stop-EflowVm

The **Stop-EflowVm** command stops the virtual machine. If the virtual machine is already stopped, no action is taken.

For more information, use the command `Get-Help Stop-EflowVm -full`.

Verify-EflowVm

The **Verify-EflowVm** command is an exposed function that checks whether the IoT Edge for Linux on Windows virtual machine was created. It takes only common parameters, and it returns **True** if the virtual machine was created and **False** if not.

For more information, use the command `Get-Help Verify-EflowVm -full`.

Next steps

Learn how to use these commands to install and provision IoT Edge for Linux on Windows in the following article:

- [Install Azure IoT Edge for Linux on Windows](#)
-

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#)

Create demo certificates to test IoT Edge device features

Article • 06/03/2024

Applies to:  IoT Edge 1.5  IoT Edge 1.4

Important

IoT Edge 1.5 LTS and IoT Edge 1.4 LTS are [supported releases](#). IoT Edge 1.4 LTS is end of life on November 12, 2024. If you are on an earlier release, see [Update IoT Edge](#).

IoT Edge devices require certificates for secure communication between the runtime, the modules, and any downstream devices. If you don't have a certificate authority to create the required certificates, you can use demo certificates to try out IoT Edge features in your test environment. This article describes the functionality of the certificate generation scripts that IoT Edge provides for testing.

Warning

These certificates expire in 30 days, and should not be used in any production scenario.

You can create certificates on any machine and then copy them over to your IoT Edge device, or generate the certificates directly on the IoT Edge device.

Prerequisites

A development machine with Git installed.

Download test certificate scripts and set up working directory

The IoT Edge repository on GitHub includes certificate generation scripts that you can use to create demo certificates. This section provides instructions for preparing the scripts to run on your computer, either on Windows or Linux.

To create demo certificates on a Windows device, you need to install OpenSSL and then clone the generation scripts and set them up to run locally in PowerShell.

Install OpenSSL

Install OpenSSL for Windows on the machine that you're using to generate the certificates. If you already have OpenSSL installed on your Windows device, ensure that openssl.exe is available in your PATH environment variable.

There are several ways to install OpenSSL, including the following options:

- **Easier:** Download and install any [third-party OpenSSL binaries](#), for example, from [OpenSSL on SourceForge](#). Add the full path to openssl.exe to your PATH environment variable.
- **Recommended:** Download the OpenSSL source code and build the binaries on your machine by yourself or via [vcpkg](#). The instructions listed below use vcpkg to download source code, compile, and install OpenSSL on your Windows machine with easy steps.
 1. Navigate to a directory where you want to install vcpkg. Follow the instructions to download and install [vcpkg](#).
 2. Once vcpkg is installed, run the following command from a PowerShell prompt to install the OpenSSL package for Windows x64. The installation typically takes about 5 minutes to complete.

```
PowerShell  
.\\vcpkg install openssl:x64-windows
```

3. Add `<vcpkg path>\installed\x64-windows\tools\openssl` to your PATH environment variable so that the openssl.exe file is available for invocation.

Prepare scripts in PowerShell

The Azure IoT Edge git repository contains scripts that you can use to generate test certificates. In this section, you clone the IoT Edge repo and execute the scripts.

1. Open a PowerShell window in administrator mode.

- Clone the IoT Edge git repo, which contains scripts to generate demo certificates. Use the `git clone` command or [download the ZIP](#).

```
PowerShell
```

```
git clone https://github.com/Azure/iotedge.git
```

- Create a directory in which you want to work and copy the certificate scripts there. All certificate and key files will be created in this directory.

```
PowerShell
```

```
mkdir wrkdir
cd .\wrkdir\
cp ..\iotedge\tools\CACertificates\*.cnf .
cp ..\iotedge\tools\CACertificates\ca-certs.ps1 .
```

If you downloaded the repo as a ZIP, then the folder name is `iotedge-master` and the rest of the path is the same.

- Enable PowerShell to run the scripts.

```
PowerShell
```

```
Set-ExecutionPolicy -ExecutionPolicy Unrestricted -Scope CurrentUser
```

- Bring the functions used by the scripts into PowerShell's global namespace.

```
PowerShell
```

```
. .\ca-certs.ps1
```

The PowerShell window will display a warning that the certificates generated by this script are only for testing purposes, and should not be used in production scenarios.

- Verify that OpenSSL has been installed correctly and make sure that there won't be name collisions with existing certificates. If there are problems, the script output should describe how to fix them on your system.

```
PowerShell
```

```
Test-CACertsPrerequisites
```

Create root CA certificate

Run this script to generate a root CA that is required for each step in this article.

The root CA certificate is used to make all the other demo certificates for testing an IoT Edge scenario. You can keep using the same root CA certificate to make demo certificates for multiple IoT Edge or downstream devices.

If you already have one root CA certificate in your working folder, don't create a new one. The new root CA certificate will overwrite the old, and any downstream certificates made from the old one will stop working. If you want multiple root CA certificates, be sure to manage them in separate folders.

Windows

1. Navigate to the working directory `wrkdir` where you placed the certificate generation scripts.
2. Create the root CA certificate and have it sign one intermediate certificate. The certificates are all placed in your working directory.

PowerShell

```
New-CACertsCertChain rsa
```

This script command creates several certificate and key files, but when articles ask for the **root CA certificate**, use the following file:

```
certs\azure-iot-test-only.root.ca.cert.pem
```

This certificate is required before you can create more certificates for your IoT Edge devices and downstream devices as described in the next sections.

Create identity certificate for the IoT Edge device

IoT Edge device identity certificates are used to provision IoT Edge devices if you choose to use X.509 certificate authentication. If you use **symmetric key** for authenticating to

IoT Hub or DPS, these certificates aren't needed, and you can skip this section.

These certificates work whether you use manual provisioning or automatic provisioning through the Azure IoT Hub Device Provisioning Service (DPS).

Device identity certificates go in the **Provisioning** section of the config file on the IoT Edge device.

Windows

1. Navigate to the working directory `wrkdir` that has the certificate generation scripts and root CA certificate.
2. Create the IoT Edge device identity certificate and private key with the following command:

PowerShell

```
New-CACertsEdgeDeviceIdentity "<device-id>"
```

The name that you pass in to this command is the device ID for the IoT Edge device in IoT Hub.

3. The new device identity command creates several certificate and key files:

 Expand table

| Type | File | Description |
|-----------------------------|---|--|
| Device identity certificate | <code>certs\iot-edge-device-identity-<device-id>.cert.pem</code> | Signed by the intermediate certificate generated earlier. Contains just the identity certificate. Specify in configuration file for DPS individual enrollment or IoT Hub provisioning. |
| Full chain certificate | <code>certs\iot-edge-device-identity-<device-id>-full-chain.cert.pem</code> | Contains the full certificate chain including the intermediate certificate. Specify in configuration file for IoT Edge to present to DPS for group enrollment provisioning. |

| Type | File | Description |
|-------------|---|--|
| Private key | <code>private\iot-edge-device-identity-<device-id>.key.pem</code> | Private key associated with the device identity certificate. Should be specified in configuration file as long as you're using some sort of certificate authentication (thumbprint or CA) for either DPS or IoT Hub. |

Create edge CA certificates

These certificates are required for **gateway scenarios** because the edge CA certificate is how the IoT Edge device verifies its identity to downstream devices. You can skip this section if you're not connecting any downstream devices to IoT Edge.

The **edge CA** certificate is also responsible for creating certificates for modules running on the device, but IoT Edge runtime can create temporary certificates if edge CA isn't configured. Edge CA certificates go in the **Edge CA** section of the `config.toml` file on the IoT Edge device. To learn more, see [Understand how Azure IoT Edge uses certificates](#).

Windows

1. Navigate to the working directory `wrkdir` that has the certificate generation scripts and root CA certificate.
2. Create the IoT Edge CA certificate and private key with the following command. Provide a name for the CA certificate. The name passed to the **New-CACertsEdgeDevice** command should *not* be the same as the hostname parameter in the config file or the device's ID in IoT Hub.

PowerShell

```
New-CACertsEdgeDevice "<CA cert name>"
```

3. This command creates several certificate and key files. The following certificate and key pair need to be copied over to an IoT Edge device and referenced in the config file:

- `certs\iot-edge-device-ca-<CA cert name>-full-chain.cert.pem`

- `private\iot-edge-device-ca-<CA cert name>.key.pem`

Create downstream device certificates

These certificates are required for setting up a downstream IoT device for a gateway scenario and want to use X.509 authentication with IoT Hub or DPS. If you want to use symmetric key authentication, you don't need to create certificates for the downstream device and can skip this section.

There are two ways to authenticate an IoT device using X.509 certificates: using self-signed certs or using certificate authority (CA) signed certs.

- For X.509 self-signed authentication, sometimes referred to as *thumbprint* authentication, you need to create new certificates to place on your IoT device. These certificates have a thumbprint in them that you share with IoT Hub for authentication.
- For X.509 certificate authority (CA) signed authentication, you need a root CA certificate registered in IoT Hub or DPS that you use to sign certificates for your IoT device. Any device using a certificate that was issued by the root CA certificate or any of its intermediate certificates can authenticate as long as the full chain is presented by the device.

The certificate generation scripts can help you make demo certificates to test out either of these authentication scenarios.

Self-signed certificates

When you authenticate an IoT device with self-signed certificates, you need to create device certificates based on the root CA certificate for your solution. Then, you retrieve a hexadecimal "thumbprint" from the certificates to provide to IoT Hub. Your IoT device also needs a copy of its device certificates so that it can authenticate with IoT Hub.

Windows

1. Navigate to the working directory `wrkdir` that has the certificate generation scripts and root CA certificate.
2. Create two certificates (primary and secondary) for the downstream device. An easy naming convention to use is to create the certificates with the name of the IoT device and then the primary or secondary label. For example:

PowerShell

```
New-CACertsDevice "<device ID>-primary"  
New-CACertsDevice "<device ID>-secondary"
```

This script command creates several certificate and key files. The following certificate and key pairs needs to be copied over to the downstream IoT device and referenced in the applications that connect to IoT Hub:

- certs\iot-device-<device ID>-primary-full-chain.cert.pem
- certs\iot-device-<device ID>-secondary-full-chain.cert.pem
- certs\iot-device-<device ID>-primary.cert.pem
- certs\iot-device-<device ID>-secondary.cert.pem
- certs\iot-device-<device ID>-primary.cert.pfx
- certs\iot-device-<device ID>-secondary.cert.pfx
- private\iot-device-<device ID>-primary.key.pem
- private\iot-device-<device ID>-secondary.key.pem

3. Retrieve the SHA1 thumbprint (called a thumbprint in IoT Hub contexts) from each certificate. The thumbprint is a 40 hexadecimal character string. Use the following openssl command to view the certificate and find the thumbprint:

PowerShell

```
Write-Host (Get-Pfxcertificate -FilePath certs\iot-device-<device  
name>-primary.cert.pem).Thumbprint
```

Run this command twice, once for the primary certificate and once for the secondary certificate. You provide thumbprints for both certificates when you register a new IoT device using self-signed X.509 certificates.

CA-signed certificates

When you authenticate an IoT device with CA-signed certificates, you need to upload the root CA certificate for your solution to IoT Hub. Use the same root CA certificate to create device certificates to put on your IoT device so that it can authenticate with IoT Hub.

The certificates in this section are for the steps in the IoT Hub X.509 certificate tutorial series. See [Understanding Public Key Cryptography and X.509 Public Key Infrastructure](#) for the introduction of this series.

1. Upload the root CA certificate file from your working directory, `certs\azure-iot-test-only.root.ca.cert.pem`, to your IoT hub.
2. If automatic verification isn't selected, use the code provided in the Azure portal to verify that you own that root CA certificate.

PowerShell

```
New-CACertsVerificationCert "<verification code>"
```

3. Create a certificate chain for your downstream device. Use the same device ID that the device is registered with in IoT Hub.

PowerShell

```
New-CACertsDevice "<device id>"
```

This script command creates several certificate and key files. The following certificate and key pairs needs to be copied over to the downstream IoT device and referenced in the applications that connect to IoT Hub:

- `certs\iot-device-<device id>.cert.pem`
- `certs\iot-device-<device id>.cert.pfx`
- `certs\iot-device-<device id>-full-chain.cert.pem`
- `private\iot-device-<device id>.key.pem`

IoT

Reference

Commands

[\[+\] Expand table](#)

| Name | Description | Type | Status |
|------------------------|---|--------------------|--------|
| az iot | Manage Internet of Things (IoT) assets. | Core and Extension | GA |

What are the Azure IoT support and help options?

Article • 02/29/2024

Here are suggestions for where you can get help when developing your Azure IoT solutions.

Create an Azure support request



Explore the range of [Azure support options and choose the plan ↗](#) that best fits, whether you're a developer just starting your cloud journey or a large organization deploying business-critical, strategic applications. Azure customers can create and manage support requests in the Azure portal.

- [Azure portal help + support ↗](#)
- [Azure portal for the United States government ↗](#)

Note

Azure IoT solutions depend on services which may have different log retention periods. To help resolve your issue, please open a support request as soon as possible to help with troubleshooting.

Post a question on Microsoft Q&A

For quick and reliable answers on your technical product questions from Microsoft Engineers, Azure Most Valuable Professionals (MVPs), or our expert community, engage with us on [Microsoft Q&A](#), Azure's preferred destination for community support.

If you can't find an answer to your problem using search, submit a new question to Microsoft Q&A. Use one of the following tags when you ask your question:

- [Azure IoT](#)
- [Azure IoT Central](#)
- [Azure IoT Edge](#)
- [Azure IoT Hub and Azure IoT Hub Device Provisioning Service \(DPS\)](#)
- [Azure IoT SDKs](#)

- [Azure Digital Twins](#)
- [Azure IoT Plug and Play](#)
- [Azure Sphere](#)
- [Azure Time Series Insights](#)
- [Azure Maps](#)

Post a question on Stack Overflow



For answers on your developer questions from the largest community developer ecosystem, ask your question on Stack Overflow.

If you do submit a new question to Stack Overflow, please use one or more of the following tags when you create the question:

- [Azure IoT Central ↗](#)
- [Azure IoT Edge ↗](#)
- [Azure IoT Hub ↗](#)
- [Azure IoT Hub Device Provisioning Service ↗](#)
- [Azure IoT SDKs ↗](#)
- [Azure Digital Twins ↗](#)
- [Azure Sphere ↗](#)
- [Azure Time Series Insights ↗](#)

Stay informed of updates and new releases



Learn about important product updates, roadmap, and announcements in [Azure Updates ↗](#).

News and information about Azure IoT is shared at the [Azure blog ↗](#) and on the [Internet of Things Show on Channel 9](#).

Also, share your experiences, engage and learn from experts in the [Internet of Things Tech Community ↗](#).

Next steps

[What is Azure IoT?](#)

Feedback

Was this page helpful?

 Yes

 No

[Provide product feedback ↗](#) | [Get help at Microsoft Q&A](#)

Glossary of IoT terms

Article • 02/28/2024

This article lists some of the common terms used in the IoT articles.

A

Advanced Message Queueing Protocol

One of the messaging protocols that [IoT Hub](#) and IoT Central support for communicating with [devices](#).

[Learn more](#)

Casing rules: Always *Advanced Message Queueing Protocol*.

First and subsequent mentions: Depending on the context spell out in full. Otherwise use the abbreviation AMQP.

Abbreviation: AMQP

Applies to: IoT Hub, IoT Central, Device developer

Allocation policy

In the [Device Provisioning Service](#), the allocation policy determines how the service assigns [devices](#) to a [Linked IoT hub](#).

Casing rules: Always lowercase.

Applies to: Device Provisioning Service

Attestation mechanism

In the [Device Provisioning Service](#), the attestation mechanism is the method used to confirm a [device](#)'s identity. The attestation mechanism is configured on an [enrollment](#).

Attestation mechanisms include X.509 certificates, Trusted Platform [Modules](#), and symmetric keys.

Casing rules: Always lowercase.

Applies to: Device Provisioning Service

Automatic deployment

A feature in [IoT Edge](#) that configures a target set of [IoT Edge devices](#) to run a set of IoT Edge [modules](#). Each deployment continuously ensures that all [devices](#) that match its [target condition](#) are running the specified set of modules, even when new devices are created or are modified to match the target condition. Each IoT Edge device only receives the highest priority deployment whose target condition it meets.

[Learn more](#)

Casing rules: Always lowercase.

Applies to: IoT Edge

Automatic device configuration

A feature of [IoT Hub](#) that enables your [solution](#) back end to assign [desired properties](#) to a set of [device twins](#) and report [device](#) status using system and custom metrics.

[Learn more](#)

Casing rules: Always lowercase.

Applies to: IoT Hub

Automatic device management

A feature of [IoT Hub](#) that automates many of the repetitive and complex tasks of managing large [device](#) fleets over the entirety of their lifecycles. The feature lets you target a set of devices based on their [properties](#), define a [desired configuration](#), and let IoT Hub update devices whenever they come into scope.

Consists of [automatic device configurations](#) and [IoT Edge automatic deployments](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Azure Digital Twins

A platform as a service (PaaS) offering for creating digital representations of real-world things, places, business processes, and people. Build twin graphs that represent entire

environments, and use them to gain insights to drive better products, optimize operations and costs, and create breakthrough customer experiences.

[Learn more](#)

Casing rules: Always capitalize when you're referring to the service.

First and subsequent mentions: When you're referring to the service, always spell out in full as *Azure Digital Twins*.

Example usage: The data in your *Azure Digital Twins* model can be routed to downstream Azure services for more analytics or storage.

Applies to: Digital Twins

Azure Digital Twins instance

A single instance of the [Azure Digital Twins](#) service in a customer's subscription. While [Azure Digital Twins](#) refers to the Azure service as a whole, your *Azure Digital Twins instance* is your individual Azure Digital Twins resource.

Casing rules: Always capitalize the service name.

First and subsequent mentions: Always spell out in full as *Azure Digital Twins instance*.

Applies to: Digital Twins

Azure IoT Explorer

A tool you can use to view the [telemetry](#) the [device](#) is sending, work with [device properties](#), and call [commands](#). You can also use the explorer to interact with and test your devices, and to manage [IoT Plug and Play devices](#).

[Learn more](#) ↗

Casing rules: Always capitalize as *Azure IoT Explorer*.

Applies to: IoT Hub, Device developer

Azure IoT Operations Preview - enabled by Azure Arc

A unified data plane for the edge. It's a collection of modular, scalable, and highly available data services that run on Azure Arc-enabled edge Kubernetes clusters. It enables data capture from various different systems and integrates with data modeling

applications such as Microsoft Fabric to help organizations deploy the industrial metaverse.

[Learn more](#)

On first mention in an article, use *Azure IoT Operations Preview - enabled by Azure Arc*. On subsequent mentions, you can use *Azure IoT Operations*. Never use an acronym.

Casing rules: Always capitalize as *Azure IoT Operations Preview - enabled by Azure Arc* or *Azure IoT Operations*.

Azure IoT Tools

A cross-platform, open-source, Visual Studio Code extension that helps you manage Azure [IoT Hub](#) and [devices](#) in VS Code. With Azure IoT Tools, IoT developers can easily develop an IoT project in VS Code

Casing rules: Always capitalize as *Azure IoT Tools*.

Applies to: IoT Hub, IoT Edge, IoT Central, Device developer

Azure IoT device SDKs

These SDKs, available for multiple languages, enable you to create [device apps](#) that interact with an [IoT hub](#) or an IoT Central application.

[Learn more](#)

Casing rules: Always refer to as *Azure IoT device SDKs*.

First and subsequent mentions: On first mention, always use *Azure IoT device SDKs*. On subsequent mentions abbreviate to *device SDKs*.

Example usage: The *Azure IoT device SDKs* are a set of device client libraries, developer guides, samples, and documentation. The *device SDKs* help you to programmatically connect devices to Azure IoT services.

Applies to: IoT Hub, IoT Central, Device developer

Azure IoT service SDKs

These SDKs, available for multiple languages, enable you to create [back-end apps](#) that interact with an [IoT hub](#).

[Learn more](#)

Casing rules: Always refer to as *Azure IoT service SDKs*.

First and subsequent mentions: On first mention, always use *Azure IoT service SDKs*. On subsequent mentions abbreviate to *service SDKs*.

Applies to: IoT Hub

B

Back-end app

In the context of [IoT Hub](#), an app that connects to one of the service-facing [endpoints](#) on an IoT hub. For example, a back-end app might retrieve [device-to-cloud](#) messages or manage the [identity registry](#). Typically, a back-end app runs in the cloud, but for simplicity many of the tutorials show back-end apps as console apps running on your local development machine.

Casing rules: Always lowercase.

Applies to: IoT Hub

Built-in endpoints

[Endpoints](#) built into [IoT Hub](#). For example, every IoT hub includes a built-in endpoint that is Event Hubs-compatible.

Casing rules: Always lowercase.

Applies to: IoT Hub

C

Cloud gateway

A cloud-hosted app that enables connectivity for [devices](#) that cannot connect directly to [IoT Hub](#) or IoT Central. A cloud [gateway](#) is hosted in the cloud in contrast to a [field gateway](#) that runs local to your devices. A common use case for a cloud gateway is to implement protocol translation for your devices.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

Cloud property

A feature in IoT Central that lets you store [device](#) metadata in the IoT Central application. Cloud [properties](#) are defined in the [device template](#), but aren't part of the [device model](#). Cloud properties are never synchronized with a device.

Casing rules: Always lowercase.

Applies to: IoT Central

Cloud-to-device

Messages sent from an [IoT hub](#) to a connected [device](#). Often, these messages are [commands](#) that instruct the device to take an action.

Casing rules: Always lowercase.

Abbreviation: Do not use *C2D*.

Applies to: IoT Hub

Command

A command is defined in an IoT Plug and Play [interface](#) to represent a method that can be called on the [digital twin](#). For example, a command to reboot a [device](#). In IoT Central, commands are defined in the [device template](#).

Applies to: IoT Hub, IoT Central, Device developer

Component

In IoT Plug and Play and [Azure Digital Twins](#), components let you build a [model interface](#) as an assembly of other interfaces. A [device model](#) can combine multiple interfaces as components. For example, a model might include a switch component and thermostat component. Multiple components in a model can also use the same interface type. For example, a model might include two thermostat components.

Casing rules: Always lowercase.

Applies to: IoT Hub, Digital Twins, Device developer

Configuration

In the context of [automatic device configuration](#) in IoT Hub, it defines the [desired configuration](#) for a set of [devices](#) twins and provides a set of metrics to report status and progress.

Casing rules: Always lowercase.

Applies to: IoT Hub

Connection string

Use in your app code to encapsulate the information required to connect to an [endpoint](#). A connection string typically includes the address of the endpoint and security information, but connection string formats vary across services. There are two types of connection string associated with the [IoT Hub](#) service:

- [Device connection strings](#) enable devices to connect to the device-facing endpoints on an IoT hub.
- [IoT Hub connection strings](#) enable [back-end apps](#) to connect to the service-facing endpoints on an IoT hub.

Casing rules: Always lowercase.

Applies to: IoT Hub, Device developer

Custom endpoints

User-defined [endpoints](#) on an [IoT hub](#) that deliver messages dispatched by a [routing rule](#). These endpoints connect directly to an event hub, a Service Bus queue, or a Service Bus topic.

Casing rules: Always lowercase.

Applies to: IoT Hub

Custom gateway

Enables connectivity for [devices](#) that cannot connect directly to [IoT Hub](#) or [IoT Central](#). You can use Azure [IoT Edge](#) to build custom [gateways](#) that implement custom logic to handle messages, custom protocol conversions, and other processing.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

D

Default component

All [IoT Plug and Play device models](#) have a default [component](#). A simple [device model](#) only has a default component - such a model is also known as a no-component [device](#). A more complex model has multiple components nested below the default component.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer

Deployment manifest

A JSON document in [IoT Edge](#) that contains the [configuration](#) data for one or more [IoT Edge device module twins](#).

Casing rules: Always lowercase.

Applies to: IoT Edge, IoT Central

Desired configuration

In the context of a [device twin](#), desired [configuration](#) refers to the complete set of [properties](#) and metadata in the [device](#) twin that should be synchronized with the device.

Casing rules: Always lowercase.

Applies to: IoT Hub

Desired properties

In the context of a [device twin](#), desired [properties](#) is a subsection of the [device](#) twin that is used with [reported properties](#) to synchronize device [configuration](#) or condition.

Desired properties can only be set by a [back-end app](#) and are observed by the [device app](#). IoT Central uses the term [writable properties](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Device

In the context of IoT, a device is typically a small-scale, standalone computing device that may collect data or control other devices. For example, a device might be an environmental monitoring device, or a controller for the watering and ventilation systems in a greenhouse.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, IoT Edge, Device Provisioning Service, Device developer

Device Provisioning Service

A helper service for [IoT Hub](#) and IoT Central that you use to configure zero-touch [device provisioning](#). With the DPS, you can provision millions of [devices](#) in a secure and scalable manner.

Casing rules: Always capitalized as *Device Provisioning Service*.

First and subsequent mentions: IoT Hub Device Provisioning Service

Abbreviation: DPS

Applies to: IoT Hub, Device Provisioning Service, IoT Central

Device REST API

A REST API you can use on a [device](#) to send [device-to-cloud](#) messages to an [IoT hub](#), and receive [cloud-to-device](#) messages from an IoT hub. Typically, you should use one of the higher-level [Azure IoT device SDKs](#).

[Learn more](#)

Casing rules: Always *device REST API*.

Applies to: IoT Hub

Device app

A [device](#) app runs on your device and handles the communication with your [IoT hub](#) or IoT Central application. Typically, you use one of the [Azure IoT device SDKs](#) when you implement a device app.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer

Device builder

The person responsible for creating the code to run on your [devices](#). Device builders typically use one of the [Azure IoT device SDKs](#) to implement the device client. A device builder uses a [device model](#) and [interfaces](#) when implementing code to run on an [IoT Plug and Play device](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, IoT Edge, Device developer

Device identity

A unique identifier assigned to every [device](#) registered in the [IoT Hub identity registry](#) or in an IoT Central application.

Casing rules: Always lowercase. If you're using the abbreviation, *ID* is all upper case.

Abbreviation: Device ID

Applies to: IoT Hub, IoT Central

Device management

[Device](#) management encompasses the full lifecycle associated with managing the devices in your IoT [solution](#) including planning, provisioning, configuring, monitoring, and retiring.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

Device model

A description, that uses the [Digital Twins Definition Language](#), of the capabilities of a [device](#). Capabilities include [telemetry](#), [properties](#), and [commands](#).

[Learn more](#)

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer, Digital Twins

Device provisioning

The process of adding the initial [device](#) data to the stores in your [solution](#). To enable a new device to connect to your hub, you must add a device ID and keys to the [IoT Hub identity registry](#). The [Device Provisioning Service](#) can automatically provision devices in an IoT hub or IoT Central application.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Device template

In IoT Central, a [device](#) template is a blueprint that defines the characteristics and behaviors of a type of device that connects to your application.

For example, the device template can define the [telemetry](#) that a device sends so that IoT Central can create visualizations that use the correct units and data types. A [device model](#) is part of the device template.

Casing rules: Always lowercase.

Abbreviation: Avoid abbreviating to *template* as IoT Central also has application templates.

Applies to: IoT Central

Device twin

A [device](#) twin is JSON document that stores device state information such as metadata, [configurations](#), and conditions. [IoT Hub](#) persists a device twin for each device that you provision in your IoT hub. Device twins enable you to synchronize device conditions and configurations between the device and the [solution](#) back end. You can query device twins to locate specific devices and for the status of long-running operations.

See also [Digital twin](#)

Casing rules: Always lowercase.

Applies to: IoT Hub

Device-to-cloud

Refers to messages sent from a connected [device](#) to [IoT Hub](#) or IoT Central.

Casing rules: Always lowercase.

Abbreviation: Do not use *D2C*.

Applies to: IoT Hub

Digital Twins Definition Language

A JSON-LD language for describing [models](#) and [interfaces](#) for [IoT Plug and Play devices](#) and [Azure Digital Twins](#) entities. The language enables the IoT platform and IoT [solutions](#) to use the semantics of the entity.

[Learn more ↗](#)

First and subsequent mentions: Spell out in full as *Digital Twins Definition Language*.

Abbreviation: DTDL

Applies to: IoT Hub, IoT Central, Digital Twins

Digital twin

A digital twin is a collection of digital data that represents a physical object. Changes in the physical object are reflected in the digital twin. In some scenarios, you can use the digital twin to manipulate the physical object. The [Azure Digital Twins service](#) uses [models](#) expressed in the [Digital Twins Definition Language](#) to represent digital twins of [physical devices](#) or higher-level abstract business concepts, enabling a wide range of cloud-based digital twin [solutions](#). An [IoT Plug and Play device](#) has a digital twin, described by a Digital Twins Definition Language [device model](#).

See also [Device twin](#)

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins, Device developer

Digital twin change events

When an [IoT Plug and Play device](#) is connected to an [IoT hub](#), the hub can use its routing capability to send notifications of [digital twin](#) changes. The IoT Central data export feature can also forward digital twin change events to other services. For example, whenever a property value changes on a [device](#), IoT Hub can send a notification to an [endpoint](#) such as an event hub.

Casing rules: Always lowercase.

Abbreviation: Always spell out in full to distinguish from other types of change event.

Applies to: IoT Hub, IoT Central

Digital twin graph

In the [Azure Digital Twins](#) service, you can connect [digital twins](#) with [relationships](#) to create knowledge graphs that digitally represent your entire physical environment. A single [Azure Digital Twins instance](#) can host many disconnected graphs, or one single interconnected graph.

Casing rules: Always lowercase.

First and subsequent mentions: Use *digital twin graph* on first mention, then use *twin graph*.

Applies to: IoT Hub

Direct method

A way to trigger a method to execute on a [device](#) by invoking an API on your [IoT hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Downstream service

A relative term describing services that receive data from the current context. For example, in the context of [Azure Digital Twins](#), Time Series Insights is a downstream service if you set up your data to flow from Azure [Digital Twins](#) into Time Series Insights.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins

E

Endpoint

A named representation of a data routing service that can receive data from other services.

An [IoT hub](#) exposes multiple endpoints that enable your apps to connect to the IoT hub. There are [device](#)-facing endpoints that enable devices to perform operations such as sending [device-to-cloud](#) messages. There are service-facing management endpoints that enable [back-end apps](#) to perform operations such as [device identity](#) management. There are service-facing [built-in endpoints](#) for reading device-to-cloud messages. You can create [custom endpoints](#) to receive device-to-cloud messages dispatched by a [routing rule](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Enrollment

In the [Device Provisioning Service](#), an enrollment is the record of individual [devices](#) or groups of devices that may register with a [linked IoT hub](#) through autoprovisioning.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Enrollment group

In the [Device Provisioning Service](#) and IoT Central, an [enrollment](#) group identifies a group of [devices](#) that share an X.509 or symmetric key [attestation mechanism](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, Device Provisioning Service, IoT Central

Event Hubs-compatible endpoint

An [IoT Hub endpoint](#) that lets you use any Event Hubs-compatible method to read [device](#) messages sent to the hub. Event Hubs-compatible methods include the [Event Hubs SDKs](#) and [Azure Stream Analytics](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Event handler

A process that's triggered by the arrival of an event. For example, you can create event handlers by adding event processing code to an Azure function, and sending data to it using [endpoints](#) and [event routing](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Event routing

The process of sending events and their data from one [device](#) or service to the [endpoint](#) of another.

In [IoT Hub](#), you can define [routing rules](#) to describe how messages should be sent. In [Azure Digital Twins](#), event routes are entities that are created for this purpose. Azure [Digital Twins](#) event routes can contain filters to limit what types of events are sent to each endpoint.

Casing rules: Always lowercase.

Applies to: IoT Hub, Digital Twins

F

Field gateway

Enables connectivity for [devices](#) that can't connect directly to [IoT Hub](#) and is typically deployed locally with your devices.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

G

Gateway

A gateway enables connectivity for [devices](#) that cannot connect directly to [IoT Hub](#). See also [field gateway](#), [cloud gateway](#), and [custom gateway](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

Gateway device

An example of a [field gateway](#). A **gateway device** can be standard IoT device or an [IoT Edge device](#).

A gateway device enables connectivity for downstream devices that cannot connect directly to [IoT Hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, IoT Edge

H

Hardware security module

Used for secure, hardware-based storage of [device secrets](#). It's the most secure form of secret storage for a device. A hardware security [module](#) can store both X.509 certificates and symmetric keys. In the [Device Provisioning Service](#), an [attestation mechanism](#) can use a hardware security module.

Casing rules: Always lowercase.

First and subsequent mentions: Spell out in full on first mention as *hardware security module*.

Abbreviation: HSM

Applies to: IoT Hub, Device developer, Device Provisioning Service

I

ID scope

A unique value assigned to a [Device Provisioning Service](#) instance when it's created.

IoT Central applications make use of DPS instances and make the ID scope available through the IoT Central UI.

Casing rules: Always use *ID scope*.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Identity registry

A built-in [component](#) of an [IoT hub](#) that stores information about the individual [devices](#) permitted to connect to the hub.

Casing rules: Always lowercase.

Applies to: IoT Hub

Individual enrollment

Identifies a single [device](#) that the [Device Provisioning Service](#) can provision to an [IoT hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, Device Provisioning Service

Interface

In IoT Plug and Play, an interface describes related capabilities that are implemented by a [IoT Plug and Play device](#) or [digital twin](#). You can reuse interfaces across different [device models](#). When an interface is used in a [device model](#), it defines a [component](#) of the device. A simple device only contains a default interface.

In [Azure Digital Twins](#), *interface* may be used to refer to the top-level code item in a [Digital Twins Definition Language](#) model definition.

Casing rules: Always lowercase.

Applies to: Device developer, Digital Twins

IoT Edge

A service and related client libraries and runtime that enables cloud-driven deployment of Azure services and [solution](#)-specific code to on-premises [devices](#). [IoT Edge devices](#) can aggregate data from other devices to perform computing and analytics before sending the data to the cloud.

[Learn more](#)

Casing rules: Always capitalize as *IoT Edge*.

First and subsequent mentions: Spell out as *Azure IoT Edge*.

Applies to: IoT Edge

IoT Edge agent

The part of the [IoT Edge runtime](#) responsible for deploying and monitoring [modules](#).

Casing rules: Always capitalize as *IoT Edge agent*.

Applies to: IoT Edge

IoT Edge device

A [device](#) that uses containerized [IoT Edge modules](#) to run Azure services, third-party services, or your own code. On the device, the [IoT Edge runtime](#) manages the modules. You can remotely monitor and manage an IoT Edge device from the cloud.

Casing rules: Always capitalize as *IoT Edge device*.

Applies to: IoT Edge

IoT Edge hub

The part of the [IoT Edge runtime](#) responsible for [module](#) to module, upstream, and downstream communications.

Casing rules: Always capitalize as *IoT Edge hub*.

Applies to: IoT Edge

IoT Edge runtime

Includes everything that Microsoft distributes to be installed on an [IoT Edge device](#). It includes Edge agent, Edge hub, and the [IoT Edge](#) security daemon.

Casing rules: Always capitalize as *IoT Edge runtime*.

Applies to: IoT Edge

IoT Hub

A fully managed Azure service that enables reliable and secure bidirectional communications between millions of [devices](#) and a [solution](#) back end. For more information, see [What is Azure IoT Hub?](#). Using your Azure subscription, you can create IoT hubs to handle your IoT messaging workloads.

[Learn more](#)

Casing rules: When referring to the service, capitalize as *IoT Hub*. When referring to an instance, capitalize as *IoT hub*.

First and subsequent mentions: Spell out in full as *Azure IoT Hub*. Subsequent mentions can be *IoT Hub*. If the context is clear, use *hub* to refer to an instance.

Example usage: The Azure IoT Hub service enables secure, bidirectional communication. The device sends data to your IoT hub.

Applies to: IoT Hub

IoT Hub Resource REST API

An API you can use to manage the [IoT hubs](#) in your Azure subscription with operations such as creating, updating, and deleting hubs.

Casing rules: Always capitalize as *IoT Hub Resource REST API*.

Applies to: IoT Hub

IoT Hub metrics

A feature in the Azure portal that lets you monitor the state of your [IoT hubs](#). IoT Hub metrics enable you to assess the overall health of an IoT hub and the [devices](#) connected to it.

Casing rules: Always capitalize as *IoT Hub metrics*.

Applies to: IoT Hub

IoT Hub query language

A SQL-like language for [IoT Hub](#) that lets you query your [jobs](#), [digital twins](#), and [device twins](#).

Casing rules: Always capitalize as *IoT Hub query language*.

First and subsequent mentions: Spell out in full as *IoT Hub query language*, if the context is clear subsequent mentions can be *query language*.

Applies to: IoT Hub

IoT Plug and Play bridge

An open-source application that enables existing sensors and peripherals attached to Windows or Linux [gateways](#) to connect as [IoT Plug and Play devices](#).

Casing rules: Always capitalize as *IoT Plug and Play bridge*.

First and subsequent mentions: Spell out in full as *IoT Plug and Play bridge*. If the context is clear, subsequent mentions can be *bridge*.

Applies to: IoT Hub, Device developer, IoT Central

IoT Plug and Play conventions

A set of conventions that IoT [devices](#) should follow when they exchange data with a [solution](#).

Casing rules: Always capitalize as *IoT Plug and Play conventions*.

Applies to: IoT Hub, IoT Central, Device developer

IoT Plug and Play device

Typically a small-scale, standalone computing [device](#) that collects data or controls other devices, and that runs software or firmware that implements a [device model](#). For example, an IoT Plug and Play device might be an environmental monitoring device, or a controller for a smart-agriculture irrigation system. An IoT Plug and Play device might be implemented directly or as an [IoT Edge module](#).

Casing rules: Always capitalize as *IoT Plug and Play device*.

Applies to: IoT Hub, IoT Central, Device developer

IoT extension for Azure CLI

An extension for the Azure CLI. The extension lets you complete tasks such as managing your [devices](#) in the [identity registry](#), sending and receiving device messages, and monitoring your [IoT hub](#) operations.

[Learn more](#)

Casing rules: Always capitalize as *IoT extension for Azure CLI*.

Applies to: IoT Hub, IoT Central, IoT Edge, Device Provisioning Service, Device developer

J

Job

In the context of [IoT Hub](#), jobs let you schedule and track activities on a set of [devices](#) registered with your IoT hub. Activities include updating [device twin desired properties](#), updating device twin [tags](#), and invoking [direct methods](#). IoT Hub also uses jobs to import to and export from the [identity registry](#).

In the context of IoT Central, jobs let you manage your connected devices in bulk by setting [properties](#) and calling [commands](#). IoT Central jobs also let you update cloud properties in bulk.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

L

Leaf device

A [device](#) with no downstream devices connected. Typically leaf devices are connected to a [gateway device](#).

Casing rules: Always lowercase.

Applies to: IoT Edge, IoT Central, Device developer

Lifecycle event

In [Azure Digital Twins](#), this type of event is fired when a data item—such as a [digital twin](#), a [relationship](#), or an [event handler](#) is created or deleted from your [Azure Digital Twins instance](#).

Casing rules: Always lowercase.

Applies to: Digital Twins, IoT Hub, IoT Central

Linked IoT hub

An [IoT hub](#) that is linked to a [Device Provisioning Service](#) instance. A DPS instance can register a [device](#) ID and set the initial [configuration](#) in the [device twins](#) in linked IoT hubs.

Casing rules: Always capitalize as *linked IoT hub*.

Applies to: IoT Hub, Device Provisioning Service

M

MQTT

One of the messaging protocols that [IoT Hub](#) and IoT Central support for communicating with [devices](#). MQTT doesn't stand for anything.

[Learn more](#)

First and subsequent mentions: MQTT

Abbreviation: MQTT

Applies to: IoT Hub, IoT Central, Device developer

Model

A definition of a type of entity in your physical environment, including its [properties](#), [telemetries](#), and [components](#). Models are used to create [digital twins](#) that represent specific physical objects of this type. Models are written in the [Digital Twins Definition Language](#).

In the [Azure Digital Twins](#) service, models define [devices](#) or higher-level abstract business concepts. In IoT Plug and Play, [device models](#) describe devices specifically.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins, Device developer

Model ID

When an [IoT Plug and Play device](#) connects to an [IoT Hub](#) or [IoT Central](#) application, it sends the [model ID](#) of the [Digital Twins Definition Language](#) model it implements. Every model has a unique model ID. This model ID enables the [solution](#) to find the [device model](#).

Casing rules: Always capitalize as *model ID*.

Applies to: IoT Hub, IoT Central, Device developer, Digital Twins

Model repository

Stores [Digital Twins Definition Language models](#) and [interfaces](#). A [solution](#) uses a [model ID](#) to retrieve a model from a repository.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins

Model repository REST API

An API for managing and interacting with a [model repository](#). For example, you can use the API to search for and retrieve [device models](#).

Casing rules: Always capitalize as *model repository REST API*.

Applies to: IoT Hub, IoT Central, Digital Twins

Module

The [IoT Hub device SDKs](#) let you instantiate modules where each one opens an independent connection to your IoT hub. This lets you use separate namespaces for different [components](#) on your device.

[Module identity](#) and [module twin](#) provide the same capabilities as [device identity](#) and [device twin](#) but at a finer granularity.

In [IoT Edge](#), a module is a Docker container that you can deploy to [IoT Edge devices](#). It performs a specific task, such as ingesting a message from a device, transforming a message, or sending a message to an IoT hub. It communicates with other modules and sends data to the [IoT Edge runtime](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Edge, Device developer

Module identity

A unique identifier assigned to every [module](#) that belongs to a [device](#). Module identities are also registered in the [identity registry](#).

The module identity details the security credentials the module uses to authenticate with the [IoT Hub](#) or, in the case of an [IoT Edge](#) module to the [IoT Edge hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Edge, Device developer

Module image

The docker image the [IoT Edge runtime](#) uses to instantiate module instances.

Casing rules: Always lowercase.

Applies to: IoT Edge

Module twin

Similar to [device twin](#), a [module](#) twin is JSON document that stores module state information such as metadata, [configurations](#), and conditions. [IoT Hub](#) persists a module twin for each [module identity](#) that you provision under a [device identity](#) in your IoT hub. Module twins enable you to synchronize module conditions and configurations between the module and the [solution](#) back end. You can query module twins to locate specific modules and query the status of long-running operations.

Casing rules: Always lowercase.

Applies to: IoT Hub

O

Ontology

In the context of [Digital Twins](#), a set of [models](#) for a particular domain, such as real estate, smart cities, IoT systems, energy grids, and more. Ontologies are often used as schemas for knowledge graphs like the ones in [Azure Digital Twins](#), because they provide a starting point based on industry standards and best practices.

[Learn more](#)

Applies to: Digital Twins

Operational technology

That hardware and software in an industrial facility that monitors and controls equipment, processes, and infrastructure.

Casing rules: Always lowercase.

Abbreviation: OT

Applies to: IoT Hub, IoT Central, IoT Edge

Operations monitoring

A feature of [IoT Hub](#) that lets you monitor the status of operations on your IoT hub in real time. IoT Hub tracks events across several categories of operations. You can opt into sending events from one or more categories to an IoT Hub [endpoint](#) for processing. You can monitor the data for errors or set up more complex processing based on data patterns.

Casing rules: Always lowercase.

Applies to: IoT Hub

P

Physical device

A real IoT [device](#) that connects to an [IoT hub](#). For convenience, many tutorials and quickstarts run IoT device code on a desktop machine rather than a physical device.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer, IoT Edge

Primary and secondary keys

When you connect to a [device](#)-facing or service-facing [endpoint](#) on an [IoT hub](#) or IoT Central application, your [connection string](#) includes key to grant you access. When you add a device to the [identity registry](#) or add a [shared access policy](#) to your hub, the service generates a primary and secondary key. Having two keys enables you to roll over

from one key to another when you update a key without losing access to the IoT hub or IoT Central application.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

Properties

In the context of a [digital twin](#), data fields defined in an [interface](#) that represent some persistent state of the digital twin. You can declare properties as read-only or writable. Read-only properties, such as serial number, are set by code running on the [IoT Plug and Play device](#) itself. Writable properties, such as an alarm threshold, are typically set from the cloud-based IoT [solution](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins, Device developer

Property change event

An event that results from a property change in a [digital twin](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins

Protocol gateway

A [gateway](#) typically deployed in the cloud to provide protocol translation services for [devices](#) connecting to an [IoT hub](#) or IoT Central application.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central

R

Registration

A record of a [device](#) in the [IoT Hub identity registry](#). You can register a device directly, or use the [Device Provisioning Service](#) to automate device registration.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Registration ID

A unique [device identity](#) in the [Device Provisioning Service](#). The [registration ID](#) may be the same value as the [device](#) identity.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service

Relationship

Used in the [Azure Digital Twins](#) service to connect [digital twins](#) into knowledge graphs that digitally represent your entire physical environment. The types of relationships that your twins can have are defined in the [Digital Twins Definition Language model](#).

Casing rules: Always lowercase.

Applies to: Digital Twins

Reported configuration

In the context of a [device twin](#), this refers to the complete set of [properties](#) and metadata in the [device](#) twin that are reported to the [solution](#) back end.

Casing rules: Always lowercase.

Applies to: IoT Hub, Device developer

Reported properties

In the context of a [device twin](#), reported [properties](#) is a subsection of the [device](#) twin. Reported properties can only be set by the device but can be read and queried by a [back-end app](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, Device developer

Retry policy

A way to handle transient errors when you connect to a cloud service.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer

Routing rule

A feature of [IoT Hub](#) used to route [device-to-cloud](#) messages to a built-in [endpoint](#) or to [custom endpoints](#) for processing by your [solution](#) back end.

Casing rules: Always lowercase.

Applies to: IoT Hub

S

SASL/PLAIN

A protocol that [Advanced Message Queueing Protocol](#) uses to transfer security tokens.

[Learn more ↗](#)

Abbreviation: SASL/PLAIN

Applies to: IoT Hub

Service REST API

A REST API you can use from the [solution](#) back end to manage your [devices](#). For example, you can use the [IoT Hub](#) service API to retrieve and update [device twin properties](#), invoke [direct methods](#), and schedule [jobs](#). Typically, you should use one of the higher-level service SDKs.

Casing rules: Always *service REST API*.

Applies to: IoT Hub, IoT Central, Device Provisioning Service, IoT Edge

Service operations endpoint

An [endpoint](#) that an administrator uses to manage service settings. For example, in the [Device Provisioning Service](#) you use the service endpoint to manage [enrollments](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, Device Provisioning Service, IoT Edge, Digital Twins

Shared access policy

A way to define the permissions granted to anyone who has a valid primary or secondary key associated with that policy. You can manage the shared access policies and keys for your hub in the portal.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Edge, Device Provisioning Service

Shared access signature

A shared access signature is a signed URI that points to one or more resources such as an [IoT hub endpoint](#). The URI includes a token that indicates how the resources can be accessed by the client. One of the query parameters, the signature, is constructed from the SAS parameters and signed with the key that was used to create the SAS. This signature is used by Azure Storage to authorize access to the storage resource.

Casing rules: Always lowercase.

Abbreviation: SAS

Applies to: IoT Hub, Digital Twins, IoT Central, IoT Edge

Simulated device

For convenience, many of the tutorials and quickstarts run [device](#) code with simulated sensors on your local development machine. In contrast, a [physical device](#) such as an MXCHIP has real sensors and connects to an [IoT hub](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device developer, IoT Edge, Digital Twins, Device Provisioning Service

Solution

In the context of IoT, *solution* typically refers to an IoT solution that includes elements such as [devices](#), [device apps](#), an [IoT hub](#), other Azure services, and [back-end apps](#).

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Device Provisioning Service, IoT Edge, Digital Twins

System properties

In the context of a [device twin](#), the read-only [properties](#) that include information regarding the [device](#) usage such as last activity time and connection state.

Casing rules: Always lowercase.

Applies to: IoT Hub

T

Tag

In the context of a [device twin](#), tags are [device](#) metadata stored and retrieved by the [solution](#) back end in the form of a JSON document. Tags are not visible to apps on a device.

Casing rules: Always lowercase.

Applies to: IoT Hub

Target condition

In an [IoT Edge](#) deployment, the target condition selects the target [devices](#) of the deployment. The target condition is continuously evaluated to include any new devices that meet the requirements or remove devices that no longer do.

Casing rules: Always lowercase.

Applies to: IoT Edge

Telemetry

The data, such as wind speed or temperature, sent to an [IoT hub](#) that was collected by a [device](#) from its sensors.

Unlike [properties](#), telemetry is not stored on a [digital twin](#); it is a stream of time-bound data events that need to be handled as they occur.

In IoT Plug and Play and [Azure Digital Twins](#), telemetry fields defined in an [interface](#) represent measurements. These measurements are typically values such as sensor readings that are sent by devices, like [IoT Plug and Play devices](#), as a stream of data.

Casing rules: Always lowercase.

Example usage: Don't use the word *telemetries*, telemetry refers to the collection of data a device sends. For example: When the device connects to your IoT hub, it starts sending telemetry. One of the telemetry values the device sends is the environmental temperature.

Applies to: IoT Hub, IoT Central, Digital Twins, IoT Edge, Device developer

Telemetry event

An event in an [IoT hub](#) that indicates the arrival of [telemetry](#) data.

Casing rules: Always lowercase.

Applies to: IoT Hub

Twin queries

A feature of [IoT Hub](#) that lets you use a SQL-like query language to retrieve information from your [device twins](#) or [module twins](#).

Casing rules: Always lowercase.

Applies to: IoT Hub

Twin synchronization

The process in [IoT Hub](#) that uses the [desired properties](#) in your [device twins](#) or [module twins](#) to configure your [devices](#) or [modules](#) and retrieve [reported properties](#) from them to store in the twin.

Casing rules: Always lowercase.

Applies to: IoT Hub

U

Upstream service

A relative term describing services that feed data into the current context. For instance, in the context of [Azure Digital Twins](#), [IoT Hub](#) is considered an upstream service because data flows from IoT Hub into Azure Digital Twins.

Casing rules: Always lowercase.

Applies to: IoT Hub, IoT Central, Digital Twins

Feedback

Was this page helpful?



[Provide product feedback](#) | [Get help at Microsoft Q&A](#)

Azure IoT Hub Documentation

Learn about IoT Hub

OVERVIEW

[What is IoT Hub?](#)

CONCEPT

[Compare IoT Hub and Event Hubs](#)

[Choose your IoT Hub tier](#)

Collect data from your devices

QUICKSTART

[Send telemetry to your hub](#)

CONCEPT

[Send and receive messages](#)

[Upload video, media, and large data files](#)

Process data with Azure services

TUTORIAL

[Route telemetry](#)

[Visualize data in Power BI](#)

CONCEPT

[Understand Azure IoT Hub message routing](#)

Manage and monitor devices remotely



QUICKSTART

[Control your devices](#)



CONCEPT

[Invoke direct methods](#)

[Use device twins](#)

[Monitor IoT Hub](#)

[Update devices using Device Update for IoT Hub](#)

Secure your IoT solution



TUTORIAL

[Create and upload certificates for testing](#)



CONCEPT

[Security best practices](#)

[Control access to IoT Hub](#)

Develop with SDKs and tools



CONCEPT

[Learn about Azure IoT SDKs](#)

[Use the Azure IoT Hub extension for VS Code ↗](#)

[Supported device protocols](#)

Azure IoT Hub Device Provisioning Service Documentation

The IoT Hub Device Provisioning Service (DPS) is a helper service for IoT Hub that enables zero-touch, just-in-time provisioning to the right IoT hub without requiring human intervention, allowing customers to provision millions of devices in a secure and scalable manner. Learn how to provision devices to your IoT hubs with our quickstarts, tutorials, and samples.

Learn about Device Provisioning Service

OVERVIEW

[What is DPS?](#)

CONCEPT

[Understand the roles and operations of auto-provisioning](#)

Get started

QUICKSTART

[Set up DPS using the Azure portal](#)

[Provision a simulated symmetric key device](#)

[Provision a simulated X.509 certificate device](#)

[Provision a simulated TPM device](#)

Manage devices for multiple IoT hubs

TUTORIAL

[Provision for geolatency](#)

[Use custom allocation policies](#)

HOW-TO GUIDE

[Link multiple IoT hubs to DPS](#)

[Use allocation policies to assign devices to IoT hubs](#)

Develop with SDKs



REFERENCE

[Libraries and SDKs](#)

Secure your IoT solution



CONCEPT

[Security practices for device manufacturers](#)



HOW-TO GUIDE

[Configure verified CA certificates](#)

[Roll X.509 device certificates](#)

[Control access to DPS](#)

Azure IoT Central documentation

Azure IoT Central is an IoT application platform (aPaaS) that simplifies the creation of IoT solutions. Azure IoT Central provides a ready-to-use UX and API surface built to connect, manage, and operate fleets of devices at scale. Learn how to use IoT Central with our quickstarts, tutorials, and REST API documentation.

Get started

OVERVIEW

[What is Azure IoT Central?](#)

[Tour of the UI](#)

[Tour of the API](#)

ARCHITECTURE

[Azure IoT Central architecture](#)

QUICKSTART

[Connect your first device](#)

Common questions

CONCEPT

[What does it mean for IoT Central to have high availability, disaster recovery, and elastic scale?](#)

[How do I extend IoT Central if it's missing something I need?](#)

[What are IoT Central's scale limits?](#)

[Where can I find demo IoT Central applications?](#)

Connect your devices

OVERVIEW

Device connectivity guide

CONCEPT

[Device implementation](#)

TRAINING

[Create and connect a device](#)

[Connect an IoT Edge device](#)

Manage your devices and analyze your data

OVERVIEW

[Device management guide](#)

TRAINING

[Dashboard examples](#)

[Create a custom dashboard to monitor devices](#)

Extend your application

OVERVIEW

[Data integration guide](#)

TRAINING

[Export data from IoT Central](#)

[Manage an application with the REST API](#)

Secure and administer your application

OVERVIEW

 TRAINING

[Add users and roles](#)

[Use organizations to manage user access](#)

Azure IoT Edge documentation

Azure IoT Edge extends IoT Hub. Analyze device data locally instead of in the cloud to send less data to the cloud, react to events quickly, and operate offline.

About IoT Edge development

OVERVIEW

[What is Azure IoT Edge?](#)

[What is Azure IoT Edge for Linux on Windows \(EFLOW\)?](#)

CONCEPT

[Understand the Azure IoT Edge runtime and its architecture](#)

[Azure IoT Edge versions and release notes](#)

[Azure IoT Edge supported systems](#)

HOW-TO GUIDE

[How an IoT Edge device can be used as a gateway](#)

Getting started using Linux devices

QUICKSTART

[Deploy your first IoT Edge module to a virtual Linux device](#)

TUTORIAL

[Develop Azure IoT Edge modules using Visual Studio Code](#)

HOW-TO GUIDE

[Install the IoT Edge runtime on Linux devices](#)

[Register and provision Linux devices at scale](#)

[Debug IoT Edge modules using Visual Studio Code](#)



REFERENCE

[IoT Edge open-source repository ↗](#)

Getting started using Windows devices



QUICKSTART

[Deploy your first IoT Edge module to a Windows device](#)



TUTORIAL

[Develop using IoT Edge for Linux on Windows](#)



HOW-TO GUIDE

[Create and provision IoT Edge for Linux on Windows](#)

[Install the IoT Edge runtime on Windows devices](#)



REFERENCE

[IoT Edge for Linux on Windows open-source repository ↗](#)

Azure Maps Documentation

Azure Maps is a set of mapping and geospatial services that enable developers and organizations to build intelligent location-based experiences for applications across many different industries and use cases. Use the Azure Maps REST APIs and Web SDK to bring maps, geocoding, location search, routing, real-time traffic, geolocation, time zone information, and weather data into your web and mobile solutions.

About Azure Maps

OVERVIEW

[What is Azure Maps?](#)

CONCEPT

[Azure Maps Coverage](#)

[Localization support](#)

[Supported map styles](#)

VIDEO

[Introduction to Azure Maps](#)

Get Started

QUICKSTART

[Create a web application using Azure Maps](#)

CONCEPT

[Choose the right pricing tier for Maps account](#)

[Authentication with Azure Maps](#)

HOW-TO GUIDE

[Manage Azure Maps accounts](#)

[Manage authentication with Azure Maps](#)

[Manage Azure Maps account pricing tier](#)

[View Azure Maps API usage metrics](#)

Developing with Web SDK

CONCEPT

[Getting started with web map control](#)

[Azure Maps web SDK code samples ↗](#)

[Web SDK supported browsers](#)

TUTORIAL

[Search for point of interest](#)

[Route to a point of interest](#)

[Create a store locator](#)

HOW-TO GUIDE

[Use the Azure Maps map control](#)

[Use the drawing tools module](#)

[Use the services module](#)

[Use the spatial IO module](#)

[Web SDK supported browsers](#)

REFERENCE

[Map control](#)

[Drawing tools module](#)

[Service module](#)

[Spatial IO module](#)

Developing with Location-Based Services

CONCEPT

Zoom levels and tile grids

TUTORIAL

[EV Routing using Azure Notebooks \(Python\)](#)

HOW-TO GUIDE

[Search for an address \(geocoding\)](#)
[Best Practices using Search Service](#)
[Best Practices using Route Service](#)
[Render custom data on static map](#)

REFERENCE

[Search Service](#)
[Route Service](#)
[Traffic Service](#)
[Time Zones Service](#)

Spatial analysis with Power BI

HOW-TO GUIDE

[Getting started](#)
[Understanding layers](#)
[Manage access](#)

Developing with Weather services

CONCEPT

[Weather services concepts](#)
[Weather services FAQ](#)

TUTORIAL

Analyze weather data using Azure Notebooks (Python)

HOW-TO GUIDE

[Request real-time and forecasted weather data](#)

REFERENCE

[Weather services](#)

Azure Time Series Insights Documentation

Learn how to run Azure IoT analytics in the cloud with fully managed event processing using quickstarts, tutorials, JavaScript samples, and REST API documentation. Analyze data from applications, sensors, devices, and more in real time.

About Azure Time Series Insights

OVERVIEW

[What is Azure Time Series Insights Gen2?](#)

[Use Cases](#)

[Client SDK sample visualization ↗](#)

CONCEPT

[Visualize data in Time Series Insights Explorer](#)

[Time Series Model](#)

VIDEO

[Analyzing Data using Time Series Insights Gen2](#)

Get started

QUICKSTART

[Explore Azure Time Series Insights Gen2](#)

TUTORIAL

[Azure Time Series Insights Gen2 tutorial](#)

DEPLOY

[Connect to Event Hubs](#)

[Connect to IoT Hub](#)

[Plan your environment](#)

[Data ingestion overview](#)

 **HOW-TO GUIDE**

[Provision and manage](#)

[How to select a TSID](#)

[Grant data access](#)

[Create Time Series Model](#)

[Query data](#)

API Reference

 **REFERENCE**

[REST APIs Overview](#)

[Data Access Concepts Gen2](#)

[JavaScript Client SDK ↗](#)

Azure IoT Hub SDKs

Article • 05/06/2024

IoT Hub provides three categories of software development kits (SDKs) to help you build device and back-end applications:

- **IoT Hub device SDKs** enable you to build applications that run on your IoT devices using the device client or module client. These apps send telemetry to your IoT hub, and optionally receive messages, jobs, methods, or twin updates from your IoT hub. You can use these SDKs to build device apps that use [Azure IoT Plug and Play](#) conventions and models to advertise their capabilities to IoT Plug and Play-enabled applications. You can also use the module client to author modules for [Azure IoT Edge](#).
- **IoT Hub service SDKs** enable you to build backend applications to manage your IoT hub, and optionally send messages, schedule jobs, invoke direct methods, or send desired property updates to your IoT devices or modules.
- **IoT Hub management SDKs** help you build backend applications that manage the IoT hubs in your Azure subscription.

Microsoft also provides a set of SDKs for provisioning devices through and building backend services for the Device Provisioning Service. To learn more, see [Microsoft SDKs for IoT Hub Device Provisioning Service](#).

Learn about the [benefits of developing using Azure IoT SDKs](#).

ⓘ Note

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier for your solution](#).

Azure IoT Hub device SDKs

The Microsoft Azure IoT device SDKs contain code that facilitates building applications that connect to and are managed by Azure IoT Hub services. These SDKs can run on a general MPU-based computing device such as a PC, tablet, smartphone, or Raspberry Pi. The SDKs support development in C and in modern managed languages including in C#, Node.js, Python, and Java.

The SDKs are available in **multiple languages** providing the flexibility to choose which best suits your team and scenario.

[+] Expand table

| Language | Package | Source | Quickstarts | Samples | Reference |
|----------|----------------------------|--------------------------|------------------------------------|---------------------------|-----------------------------|
| .NET | NuGet ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference |
| Python | pip ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference |
| Node.js | npm ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference |
| Java | Maven ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference |
| C | packages ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference ↗ |

The Java device SDK includes [samples for Android ↗](#).

The C device SDK includes [samples for iOS that use CocoaPods ↗](#).

⚠ Warning

The **Azure IoT C SDK** isn't suitable for embedded applications due to its memory management and threading model. For embedded device SDK options, see the [embedded device SDKs](#).

Learn more about the IoT Hub device SDKs in the [IoT device development documentation](#).

Embedded device SDKs

These SDKs were designed and created to run on devices with limited compute and memory resources and are implemented using the C language.

The embedded device SDKs are available for **multiple operating systems** providing the flexibility to choose which best suits your scenario.

[+] Expand table

| RTOS | SDK | Source | Samples | Reference |
|-----------------|-----------------------|--------------------------|-----------------------------|-----------------------------|
| Eclipse ThreadX | Azure RTOS Middleware | GitHub ↗ | Quickstarts | Reference ↗ |
| FreeRTOS | FreeRTOS Middleware | GitHub ↗ | Samples ↗ | Reference ↗ |

| RTOS | SDK | Source | Samples | Reference |
|------------|--------------------------|--------------------------|---------------------------|-----------------------------|
| Bare Metal | Azure SDK for Embedded C | GitHub ↗ | Samples ↗ | Reference ↗ |

Azure IoT Hub service SDKs

The Azure IoT service SDKs contain code to facilitate building applications that interact directly with IoT Hub to manage devices and security.

[Expand table](#)

| Platform | Package | Code Repository | Samples | Reference |
|----------|-------------------------|--------------------------|---------------------------|---------------------------|
| .NET | NuGet ↗ | GitHub ↗ | Samples ↗ | Reference |
| Java | Maven ↗ | GitHub ↗ | Samples ↗ | Reference |
| Node | npm ↗ | GitHub ↗ | Samples ↗ | Reference |
| Python | pip ↗ | GitHub ↗ | Samples ↗ | Reference |

Azure IoT Hub management SDKs

The IoT Hub management SDKs help you build backend applications that manage the IoT hubs in your Azure subscription.

[Expand table](#)

| Platform | Package | Code repository | Reference |
|----------|-------------------------|--------------------------|---------------------------|
| .NET | NuGet ↗ | GitHub ↗ | Reference |
| Java | Maven ↗ | GitHub ↗ | Reference |
| Node.js | npm ↗ | GitHub ↗ | Reference |
| Python | pip ↗ | GitHub ↗ | Reference |

SDKs for related Azure IoT services

Azure IoT SDKs are also available for the following services:

- [SDKs for IoT Hub Device Provisioning Service](#): To help you provision devices through and build backend services for the Device Provisioning Service.

- [SDKs for Device Update for IoT Hub](#): To help you deploy over-the-air (OTA) updates for IoT devices.

Next steps

Learn how to [manage connectivity and reliable messaging](#) using the IoT Hub device SDKs.

Azure IoT Hub SDKs

Article • 05/06/2024

IoT Hub provides three categories of software development kits (SDKs) to help you build device and back-end applications:

- **IoT Hub device SDKs** enable you to build applications that run on your IoT devices using the device client or module client. These apps send telemetry to your IoT hub, and optionally receive messages, jobs, methods, or twin updates from your IoT hub. You can use these SDKs to build device apps that use [Azure IoT Plug and Play](#) conventions and models to advertise their capabilities to IoT Plug and Play-enabled applications. You can also use the module client to author modules for [Azure IoT Edge](#).
- **IoT Hub service SDKs** enable you to build backend applications to manage your IoT hub, and optionally send messages, schedule jobs, invoke direct methods, or send desired property updates to your IoT devices or modules.
- **IoT Hub management SDKs** help you build backend applications that manage the IoT hubs in your Azure subscription.

Microsoft also provides a set of SDKs for provisioning devices through and building backend services for the Device Provisioning Service. To learn more, see [Microsoft SDKs for IoT Hub Device Provisioning Service](#).

Learn about the [benefits of developing using Azure IoT SDKs](#).

ⓘ Note

Some of the features mentioned in this article, like cloud-to-device messaging, device twins, and device management, are only available in the standard tier of IoT Hub. For more information about the basic and standard/free IoT Hub tiers, see [Choose the right IoT Hub tier for your solution](#).

Azure IoT Hub device SDKs

The Microsoft Azure IoT device SDKs contain code that facilitates building applications that connect to and are managed by Azure IoT Hub services. These SDKs can run on a general MPU-based computing device such as a PC, tablet, smartphone, or Raspberry Pi. The SDKs support development in C and in modern managed languages including in C#, Node.js, Python, and Java.

The SDKs are available in **multiple languages** providing the flexibility to choose which best suits your team and scenario.

[+] Expand table

| Language | Package | Source | Quickstarts | Samples | Reference |
|----------|----------------------------|--------------------------|------------------------------------|---------------------------|-----------------------------|
| .NET | NuGet ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference |
| Python | pip ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference |
| Node.js | npm ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference |
| Java | Maven ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference |
| C | packages ↗ | GitHub ↗ | Connect to IoT Hub | Samples ↗ | Reference ↗ |

The Java device SDK includes [samples for Android ↗](#).

The C device SDK includes [samples for iOS that use CocoaPods ↗](#).

⚠ Warning

The **Azure IoT C SDK** isn't suitable for embedded applications due to its memory management and threading model. For embedded device SDK options, see the [embedded device SDKs](#).

Learn more about the IoT Hub device SDKs in the [IoT device development documentation](#).

Embedded device SDKs

These SDKs were designed and created to run on devices with limited compute and memory resources and are implemented using the C language.

The embedded device SDKs are available for **multiple operating systems** providing the flexibility to choose which best suits your scenario.

[+] Expand table

| RTOS | SDK | Source | Samples | Reference |
|-----------------|-----------------------|--------------------------|-----------------------------|-----------------------------|
| Eclipse ThreadX | Azure RTOS Middleware | GitHub ↗ | Quickstarts | Reference ↗ |
| FreeRTOS | FreeRTOS Middleware | GitHub ↗ | Samples ↗ | Reference ↗ |

| RTOS | SDK | Source | Samples | Reference |
|------------|--------------------------|--------------------------|---------------------------|-----------------------------|
| Bare Metal | Azure SDK for Embedded C | GitHub ↗ | Samples ↗ | Reference ↗ |

Azure IoT Hub service SDKs

The Azure IoT service SDKs contain code to facilitate building applications that interact directly with IoT Hub to manage devices and security.

[Expand table](#)

| Platform | Package | Code Repository | Samples | Reference |
|----------|-------------------------|--------------------------|---------------------------|---------------------------|
| .NET | NuGet ↗ | GitHub ↗ | Samples ↗ | Reference |
| Java | Maven ↗ | GitHub ↗ | Samples ↗ | Reference |
| Node | npm ↗ | GitHub ↗ | Samples ↗ | Reference |
| Python | pip ↗ | GitHub ↗ | Samples ↗ | Reference |

Azure IoT Hub management SDKs

The IoT Hub management SDKs help you build backend applications that manage the IoT hubs in your Azure subscription.

[Expand table](#)

| Platform | Package | Code repository | Reference |
|----------|-------------------------|--------------------------|---------------------------|
| .NET | NuGet ↗ | GitHub ↗ | Reference |
| Java | Maven ↗ | GitHub ↗ | Reference |
| Node.js | npm ↗ | GitHub ↗ | Reference |
| Python | pip ↗ | GitHub ↗ | Reference |

SDKs for related Azure IoT services

Azure IoT SDKs are also available for the following services:

- [SDKs for IoT Hub Device Provisioning Service](#): To help you provision devices through and build backend services for the Device Provisioning Service.

- [SDKs for Device Update for IoT Hub](#): To help you deploy over-the-air (OTA) updates for IoT devices.

Next steps

Learn how to [manage connectivity and reliable messaging](#) using the IoT Hub device SDKs.