

Machine Learning and IoT Edge

A Walkthrough – **DRAFT: WORK IN PROGRESS**

1 Introduction

Frequently, IoT applications want to take advantage of the intelligent cloud and the intelligent edge. In this document, we walk you through training a machine learning model with data collected from IoT devices in the cloud, deploying that model to IoT Edge, and maintaining and refining the model periodically.

The primary objective of this document is to introduce processing of IoT data with machine learning (ML), specifically on the edge. While we will touch many aspects of a general ML workflow, this walk-through is not intended as an in-depth introduction to machine learning. As a case in point, we do not attempt to create a highly optimized model for the use case – we just want to illustrate the process of creating and using a viable model for IoT data processing.

1.1 Walk-through Use Case: Predictive Maintenance

For this walk-through, we are using the use case from a **NASA predictive maintenance competition** published in 2008. Goal is to predict remaining useful life (RUL) of a set of turbofan airplane engines. The data for the scenario is synthetic and relatively compact. It is available here:

<https://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-repository/#turbofan>

From the readme file:

Experimental Scenario

Data sets consists of multiple multivariate time series. Each data set is further divided into training and test subsets. Each time series is from a different engine – i.e., the data can be considered to be from a fleet of engines of the same type. Each engine starts with different degrees of initial wear and manufacturing variation which is unknown to the user. This wear and variation is considered normal, i.e., it is not considered a fault condition. There are three operational settings that have a substantial effect on engine performance. These settings are also included in the data. The data is contaminated with sensor noise.

The engine is operating normally at the start of each time series and develops a fault at some point during the series. In the training set, the fault grows in magnitude until system failure. In the test set, the time series ends some time prior to system failure. The objective of the competition is to predict the number of remaining operational cycles before failure in the test set, i.e., the number of operational cycles after the last cycle that the engine will continue to operate. Also provided a vector of true Remaining Useful Life (RUL) values for the test data.

Because the data was published for a competition, several approaches to derive machine learning models have been published independently. We found that studying examples is helpful in

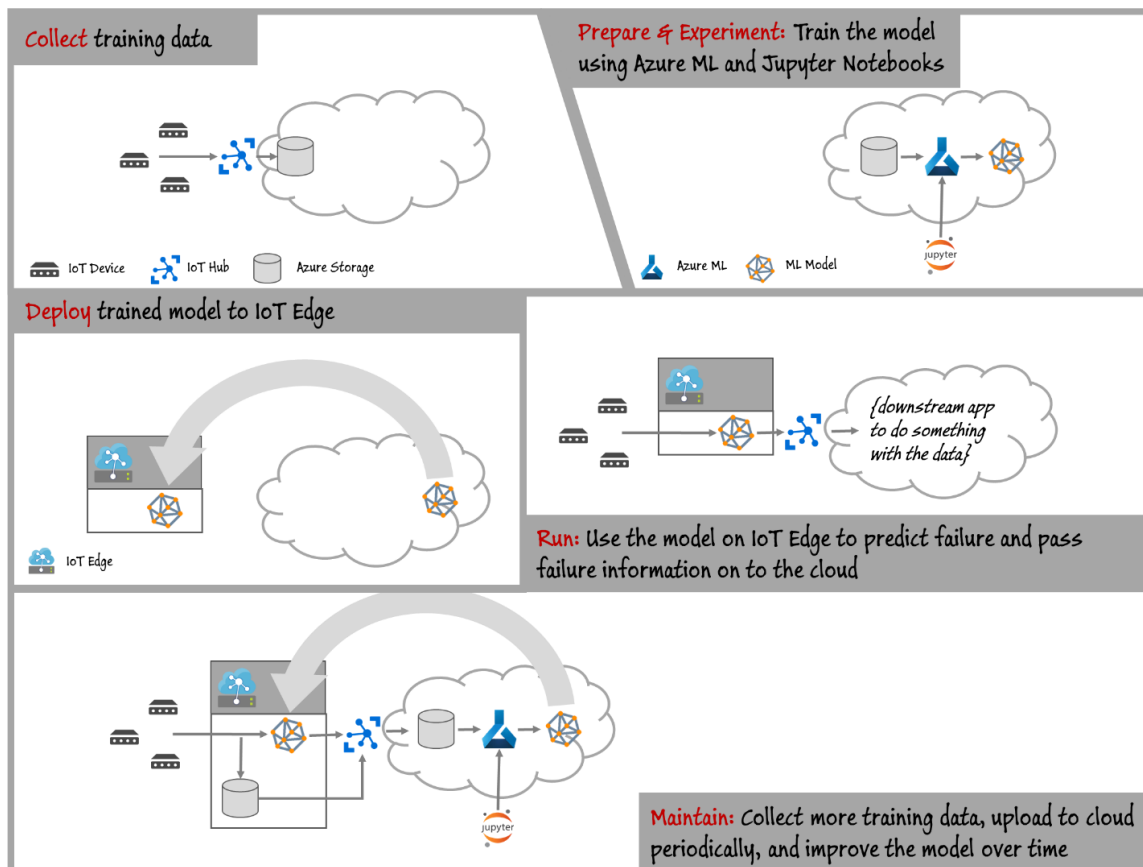
understanding the process and reasoning involved in the creation of a specific ML model. See for example:

https://github.com/jancervenka/turbofan_failure

<https://github.com/hankroark/Turbofan-Engine-Degradation>

1.2 Process

The picture below illustrates the rough steps we follow in this walk-through:



- **Collection of training data:** The process begins with the collection of training data. In some cases, data has already been collected and is available in a data base, or in form of data files. In other cases, especially for IoT scenarios, the data needs to be collected from IoT devices and sensors and stored in the cloud.
As we assume that you do not have a collection of turbofan engines, we provide a simple device simulator harness that sends the NASA device data to the cloud.
- **Data Preparation.** In most cases, the raw data as collected from devices and sensors will require preparation for machine learning. This may involve data clean up, data reformatting, or preprocessing to inject additional information machine learning can key off.

In the case of the airplane engine machine data, it involves calculating explicit time-to-failure times for every data point in the sample based on the actual observations on the data. This information allows the machine learning algorithm to find correlations between actual sensor data patterns and the expected remaining life time of the engine. Of course, this step is highly domain specific.

- **Experimentation.** Based on the prepared data, we can now experiment with different machine learning algorithms and parameterizations to train models and compare the results relative to one another. In this case, for testing we compare the predicted outcome computed by the model with the real outcome observed on a set of engines.

In Azure ML, we can manage the different iterations of models we create in a model registry.

- **Deployment.** Once we have a model that satisfies our success criteria, we can move to deployment. That involves wrapping the model into a web service app that can be fed with data using REST calls and return analysis results. The web service app is then packaged into a docker container, which in turn can be deployed either in the cloud or as an IoT Edge module. In this example, we focus on deployment to IoT Edge.
- **Refinement.** Our work is not done once the model is deployed. In many cases, we want to continue collecting data, now on the edge device, and periodically upload that data to the cloud. We can then use this data to re-train and refine our model, which we then can re-deploy to IoT Edge.

In this document, we will use the following set of tools:

- Azure IoT Hub for data capture
- Azure Notebooks as our main front-end for data preparation and ML experimentation. Running python code in a notebook on a subset of the sample data is a great way to get fast iterative and interactive turnaround during data preparation. Jupyter notebooks can also be used to prepare scripts to run at scale in a compute backend.
- Azure ML as a backend for ML at scale and for ML image generation. We drive the Azure ML backend using scripts prepared and tested in Jupyter notebooks.
- Azure IoT Edge for off-cloud application of a machine learning image

Obviously, there are other options available. In certain scenarios, for example, IoT Central can be used as a no-code alternative to capture initial training data from IoT devices.

2 Prerequisites

To complete the walk-through you will need access to an Azure subscription in which you have rights to create resources. Several of the services used to run this walk-through will incur Azure charges.

If you do not already have an Azure subscription you may be able to get started with an [Azure Free Account](#).

3 Setting Up a Development Machine

3.1 Introduction

Over the course of this walk-through we will be performing various developer tasks including coding, compiling, configuring and deploying IoT Edge module and IoT devices.

To provide a common base for these tasks we recommend the use of an Azure virtual machine configured specifically as your development machine for this walk-through. The VM that we create needs to be able to handle nested virtualization, which is why we chose a DS8V3 machine size.

The development VM will be set up with:

- Windows 10
- [Chocolatey](#)
- [Docker Desktop for Windows](#)
- [Git for Windows](#)
- [Git Credential Manager for Windows](#)
- [.Net Core SDK](#)
- [Python](#)
- [Visual Studio Code](#)
- [Azure PowerShell](#)
- [VS Code Extensions](#)
 - [Azure IoT Tools](#)
 - [Python](#)
 - [Azure Account](#)
 - [C#](#)
 - [Docker](#)
 - [PowerShell](#)

The developer VM is not strictly necessary – all the development tools can of course also be run on a local machine. However, we strongly recommend using the VM to ensure a level playing field.

This section will take about 30 minutes to complete.

3.2 Get the code

Clone or download the code from <https://github.com/Azure-Samples/IoTEdgeAndMISample>.

3.3 Create an Azure Virtual Machine

The DevVM directory contains the files needed to create an Azure virtual machine appropriate for completing this walk-through. To create a VM:

1. Open Powershell as an administrator and navigate to the directory where you downloaded the code. We will refer to the root directory for your source as <srcdir>.

```
cd <srcdir>\IoTEdgeAndMISample\DevVM
```

2. Allow execution of scripts

```
Set-ExecutionPolicy Bypass -Scope Process
```

```
PS C:\source\IoTEdgeAndMlSample\DevVM> Set-ExecutionPolicy Bypass -Scope Process

Execution Policy Change
The execution policy helps protect you from scripts that you do not trust. Changing the execution policy might
want to change the execution policy?
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): a
PS C:\source\IoTEdgeAndMlSample\DevVM> .\Create-AzureDevVm.ps1
```

3. Run Create-AzureDevVm.ps1 from this directory

```
.\Create-AzureDevVm.ps1
```

- When prompted provide:
 - Azure Subscription ID: found in the Azure Portal
 - Resource Group Name: memorable name for grouping the resources for your walk-through
 - Location: Azure location where the virtual machine will be created (e.g. West US 2, North Europe, see full [list](#))
 - AdminUsername: the name for the admin account you want to create and use on the virtual machine
 - AdminPassword: the password to set for the AdminUsername on the VM
- If you do not have Azure PowerShell installed, the script will install [Azure PowerShell Az module](#)
- You will be prompted to login to Azure.
- The script confirms the information for the creation of your VM. Press 'y' or 'Enter' to continue.

```

cmdlet Create-AzureDevVm.ps1 at command pipeline position 1
Supply values for the following parameters:
SubscriptionId: [1679802-8a07-48aa-8f94-44d8c9d120a]
ResourceGroupName: m1EdgeIoTWalkthrough
Location: West Us 2
AdminUsername: mmydland
AdminPassword: *****
GitHubUsername: mmydland
NuGet provider is required to continue
PowerShellGet requires NuGet provider version '2.8.5.201' or newer to interact with NuGet-based repositories. The NuGet provider must
'C:\Users\markmyd\AppData\Local\PackageManagement\ProviderAssemblies'. You can also install the NuGet provider by running 'Install-Pac
provider now?
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): y

Untrusted repository
You are installing the modules from an untrusted repository. If you trust this repository, change its InstallationPolicy value by runn
[Y] Yes [A] Yes to All [N] No [L] No to All [S] Suspend [?] Help (default is "N"): y
Enable alias Az PowerShell
Please login to Azure...
WARNING: To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code BH8UN3G4H to authenticate
Selecting subscription [1679802-8a07-48aa-8f94-44d8c9d120a]

You are about to create a virtual machine in Azure:
- Subscription [1679802-8a07-48aa-8f94-44d8c9d120a]
- Resource group m1EdgeIoTWalkthrough
- Location 'West Us 2'

Are you sure you want to continue?
[Y] Yes [N] No (default is "Y"):

Starting deployment of the demo VM which may take a while.
Progress can be monitored from the Azure Portal (http://portal.azure.com).
1. Find the resource group m1EdgeIoTWalkthrough in [1679802-8a07-48aa-8f94-44d8c9d120a] subscription.
2. In the Deployments page open deployment IotEdgeM1DemoVm-PdClb.

Enabling Hyper-V on Azure VM...

Installing Chocolatey on Azure VM...

Installing necessary software Azure VM...

Restarting the VM...
WARNING: Breaking changes in the cmdlet 'Restart-AzVM':
WARNING: - The parameter : 'Name' is changing.

WARNING: Change description : Name will be removed from the Id parameter sets in an upcoming breaking change release.

Writing the VM RDP file to C:\Users\markmyd\Desktop\IoTMLDemo-wzps1.rdp

The VM is ready.
Visit the Azure Portal (http://portal.azure.com).
- Virtual machine name: IoTMLDemo-wzps1
- Resource group: m1EdgeIoTWalkthrough
- Subscription: [1679802-8a07-48aa-8f94-44d8c9d120a]

Use the RDP file: C:\Users\markmyd\Desktop\IoTMLDemo-wzps1.rdp to connect to the virtual machine.

WARNING: Please note this VM was configured with a shutdown schedule. Review it on the VM blade to confirm the settings work for you.

```

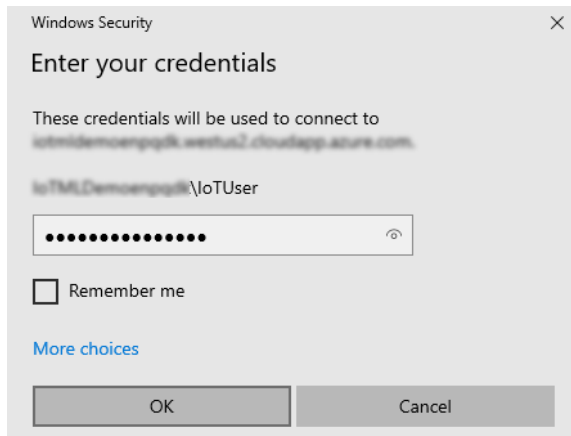
4. The script will run for several minutes as it executes the steps:
 - Create the Resource Group if it does not exist
 - Deploy the virtual machine
 - Enable Hyper-V on the VM
 - Install software need for development and clone the sample repository
 - Restart the VM
 - Create an RDP file on your desktop for connecting to the VM

3.4 Set auto-shutdown schedule

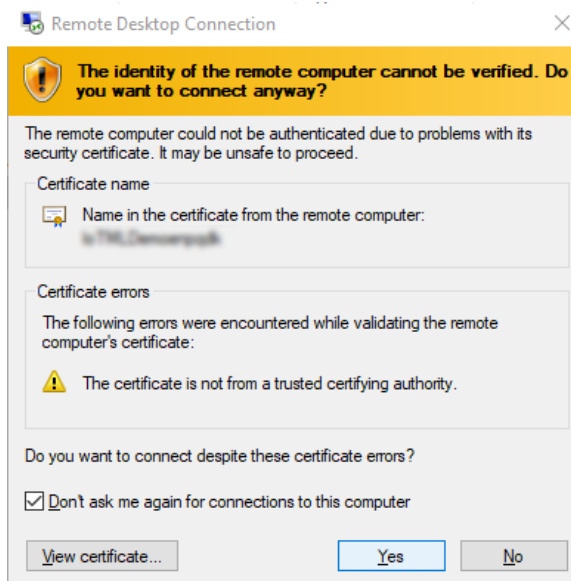
The VM has been created with an automatic shutdown schedule which is set to 1900 PST. You may need to update this timing depending on your location and schedule. To update the shutdown schedule:

1. Login into the Azure portal <http://portal.azure.com>
2. Navigate to your VM

4. When prompted, provide the AdminPassword that you used when running the script to set up the VM and click “OK”



5. You will be prompted to accept the certificate for the VM. Click the “Don’t ask me again for connections to this computer” and choose “Yes”



3.5.2 Install Visual Studio Code extensions

We add some useful extensions to Visual Studio Code to make the development experience easier.

1. In a PowerShell window navigate to C:\source\IoTEdgeAndMISample\DevVM
2. Allow scripts to be executed on the virtual machine by typing

```
Set-ExecutionPolicy Bypass -Scope CurrentUser -Force
```

3. Run the script

```
.\Enable-CodeExtensions.ps1
```

4. The script will run for a few minutes installing VS code extensions:
 - Azure IoT Tools
 - Python
 - C#

- Docker
- PowerShell

3.6 Summary

In this section we created an Azure Virtual Machine and configured it with the tools and software that will be needed to complete the remainder of the walk-through. In the next section we will begin creating an IoT solution by setting up an Azure IoT Hub.

4 Add IoT Hub and Configure a Storage Route

4.1 Introduction

Azure IoT Hub is the heart of any IoT application. It handles secure communication between IOT devices and the cloud. It is the main coordination point for the operation of the Edge ML solution.

- IoT Hub uses routes to direct incoming data from IoT devices to other downstream services. In this tutorial, we will route data to Azure Storage so that it can be consumed by Azure Machine Learning to train our remaining useful life (RUL) classifier.
- Later, we will also use the IoT Hub to configure and deploy our Azure IoT Edge device and to manage which Edge modules are deployed to the device and how they are configured.

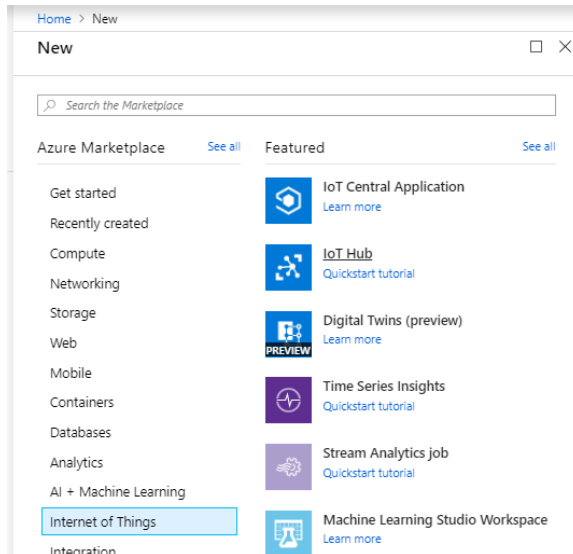
In this section, we will create an Azure IoT Hub and configure a route that will forward data received by the hub to an Azure Storage Blob container. It will take about 10 minutes to complete.

CAUTION: MAKE SURE TO DEPLOY BOTH RESOURCES, STORAGE AND IoT HUB, TO THE SAME AZURE LOCATION OR PROBLEMS WILL BE ENCOUNTERED.

4.2 Create IoT Hub

1. Navigate to <http://portal.azure.com>. You can perform all tasks in Azure portal from any web browser, but we will assume that you are logged in to your development VM.
2. From the side navigator of the portal click the “+ Create a resource” button
3. In the “New” dialog, click on “Internet of Things”

4. Under “Featured” click on “IoT Hub”



5. In the “Basics” tab of the IoT Hub fill in your Subscription, Resource Group, and Region (all should be the same as the ones used to create your VM above) and then give your IoT Hub a name

[Basics](#) [Size and scale](#) [Review + create](#)

Create an IoT Hub to help you connect, monitor, and manage billions of your IoT assets. [Learn More](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription ⓘ

* Resource Group ⓘ [Create new](#)

* Region ⓘ

* IoT Hub Name ⓘ ✓

6. Click the “Review and Create” button. Validate that your hub will be S1 for scale tier, then click “Create”. While there is a free tier option for IoT Hub, the amount of data required for this tutorial exceeds the limits of the free ties.

[Basics](#) [Size and scale](#) [Review + create](#)

BASICS

Subscription ⓘ

Resource Group ⓘ

Region ⓘ

IoT Hub Name ⓘ

SIZE AND SCALE

Pricing and scale tier ⓘ S1

Number of S1 IoT Hub units ⓘ 1

Messages per day ⓘ 400,000

Cost per month

7. You will be taken to the deployment page while the IoT Hub is created. There is no need to wait for the deployment to complete. Instead, continue on to creating your storage account.

4.3 Create Azure Storage account

As stated above, we need a storage location to route data collected through the IoT Hub to be stored so that we can access that data when training our RUL classifier. In this section, we will create the Azure Storage account that will be used for storing this data

1. Navigate to <http://portal.azure.com>
2. From the side navigator of the portal click the “+ Create a resource” button
3. In the “New” dialog, click on “Storage”
4. Under “Featured” click on “Storage account – blob, file, table, queue”
5. In the basics section of the create storage dialog enter the same subscription, resource group name, and location as were used in creating your IoT Hub. Give the storage account a unique name.

The screenshot shows the 'Create storage account' dialog in the Azure portal, with the 'Basics' tab selected. At the top, there are tabs for 'Basics', 'Advanced', 'Tags', and 'Review + create'. Below this is a descriptive paragraph about Azure Storage. The 'PROJECT DETAILS' section contains two dropdown menus: 'Subscription' (set to 'Microsoft Azure') and 'Resource group' (set to 'IoTEdgeMIIWalkthrough'), with a 'Create new' link below the resource group dropdown. The 'INSTANCE DETAILS' section includes a note about the default deployment model. Below this are five configuration rows: 'Storage account name' (text input 'turbodevicedata' with a green checkmark), 'Location' (dropdown 'West US 2'), 'Performance' (radio buttons for 'Standard' (selected) and 'Premium'), 'Account kind' (dropdown 'Storage (general purpose v1)'), and 'Replication' (dropdown 'Locally-redundant storage (LRS)').

Basics Advanced Tags Review + create

Azure Storage is a Microsoft-managed service providing cloud storage that is highly available, secure, durable, scalable, and redundant. Azure Storage includes Azure Blobs (objects), Azure Data Lake Storage Gen2, Azure Files, Azure Queues, and Azure Tables. The cost of your storage account depends on the usage and the options you choose below. [Learn more](#)

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

* Subscription

* Resource group [Create new](#)

INSTANCE DETAILS

The default deployment model is Resource Manager, which supports the latest Azure features. You may choose to deploy using the classic deployment model instead. [Choose classic deployment model](#)

* Storage account name ✓

* Location

Performance ☒ Standard ☐ Premium

Account kind

Replication

- Click on “Review + create”. Validate your settings and click “Create”
Create storage account

✓ Validation passed

Basics Advanced Tags **Review + create**

BASICS

Subscription	997,888,346,111
Resource group	IoTEdgeMIWalkthrough
Location	West US 2
Storage account name	turbodevicedata
Deployment model	Resource manager
Account kind	Storage (general purpose v1)
Replication	Locally-redundant storage (LRS)
Performance	Standard

ADVANCED

Secure transfer required	Enabled
Allow access from	All networks
Hierarchical namespace	Disabled


- You will be taken to the deployment status page. When the deployment completes click on “Go to resource” to open the storage account.


4.4 Create storage container


Within the storage account we need a specific blob container to which we will flow data from the IoT Hub. In this step we will create the blob container.

- Navigate to the storage account created in the previous step
- Under “Services” click on “Blobs”

Services

**Blobs**
REST-based object storage for unstructured data
[Explore data using Azure AD preview](#)
[Learn more](#)

**Files**
File shares that use the standard SMB 3.0 protocol
[Learn more](#)

**Tables**
Tabular data storage
[Learn more](#)

- Click on “+ Container”
- Name your container “devicedata” and click “OK”

+ Container Refresh Delete

New container

* Name
 ✓

Public access level ⓘ

OK Cancel

5. Note that the container has been added to your storage account

The screenshot shows the Azure Storage account interface. At the top, there are buttons for '+ Container', 'Refresh', and 'Delete'. Below this, it says 'Storage account: turbodevicedata'. There is a search bar with the placeholder text 'Search containers by prefix'. Below the search bar, there is a table with one row containing the name 'devicedata'.

NAME
devicedata

4.5 Add route to storage in IoT Hub

Now that we have a storage location, we need to set up a route in IoT Hub to send the messages to the container. Routes consist of a query expression that is applied to incoming messages by IoT Hub. If a message matches the expression, the data is sent along that route. Routes also have an associated endpoint, which can be Azure storage, or another downstream service to further process the data.

1. Navigate to the IoT Hub in the portal and choose “Message routing” from the side menu
2. Under Routes click the “+ Add” button


The screenshot shows the IoT Hub Message routing interface. At the top, there is a description: 'Send data from your devices to endpoints that you choose.' Below this, there are tabs for 'Routes' and 'Custom endpoints'. A note says: 'Create an endpoint, and then add a route (you can add up to 100 from each IoT hub). Messages that don't match a query are automatically sent to messages/e'. There is a button 'Disable fallback route'. Below this, there are buttons for '+ Add', 'Test all routes', and 'Delete'. At the bottom, there is a table with columns: NAME, DATA SOURCE, ROUTING QUERY, and ENDPOINT. The table currently shows 'No results'.

NAME	DATA SOURCE	ROUTING QUERY	ENDPOINT
No results			

3. In the “Add a route page”, give the route a name like “turbofanDeviceDataToStorage”
4. Click on “+ Add” next to the 'Endpoint' field and choose “Blob storage”
5. In the “Add a route” page, click the “+ Add” next to the 'Endpoint' field and choose blob storage

The screenshot shows the 'Add a route' page in the IoT Hub portal. It has a title 'Add a route' with a plus icon. There are four fields: 'Name' (with a red asterisk and an info icon), 'Endpoint' (with a red asterisk and an info icon), 'Data source' (with a red asterisk and an info icon), and 'Enable route' (with a red asterisk and an info icon). The 'Name' field contains 'turbofanDeviceDataToStorage' and has a green checkmark. The 'Endpoint' field is empty and has a '+ Add' button next to it. The 'Data source' field contains 'Device Telemetry Messages'. The 'Enable route' field has 'Enable' and 'Disable' buttons. A dropdown menu is open next to the '+ Add' button, showing options: 'Event hubs', 'Service bus queue', 'Service bus topic', and 'Blob storage' (which is highlighted).

- This brings up the “Add a storage endpoint” page. Give the endpoint a name like “turbofanDeviceStorage” then click “Pick a container”

 **Add a storage endpoint**

Route your telemetry and device messages to Azure Storage as blobs.

* Endpoint name ⓘ

Azure Storage account and container

Create a new container, or choose an existing one that shares a subscript

Azure Storage container

[Pick a container](#)

- Select the storage account created above

[+ Storage account](#) [Refresh](#)

NAME

turbofandevicedata

- In the “Containers” page select the “devicedata” you created and click on “Select”

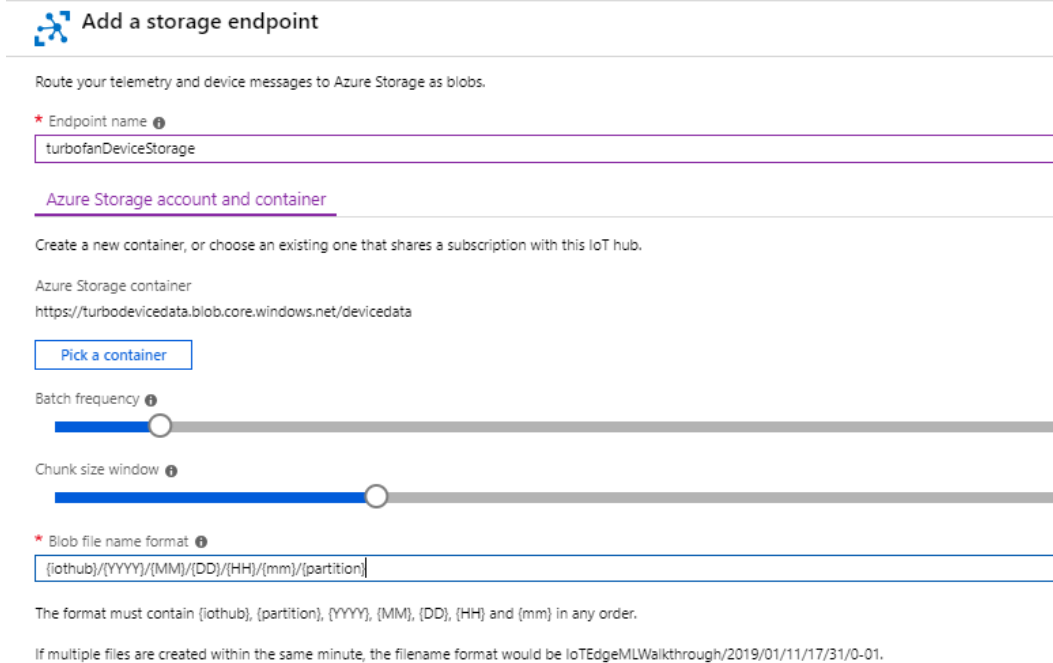
Containers
turbofandevicedata

[+ Container](#) [Refresh](#)

NAME

devicedata

9. Back in the “Add a storage endpoint” page, set the “Blob file name format” to "{iothub}/{YYYY}/{MM}/{DD}/{HH}/{mm}/{partition}" and click “Create”.



Add a storage endpoint

Route your telemetry and device messages to Azure Storage as blobs.

* Endpoint name ⓘ
turbofanDeviceStorage

Azure Storage account and container

Create a new container, or choose an existing one that shares a subscription with this IoT hub.

Azure Storage container
https://turbodevicedata.blob.core.windows.net/devicedata

[Pick a container](#)

Batch frequency ⓘ
[Slider bar]

Chunk size window ⓘ
[Slider bar]

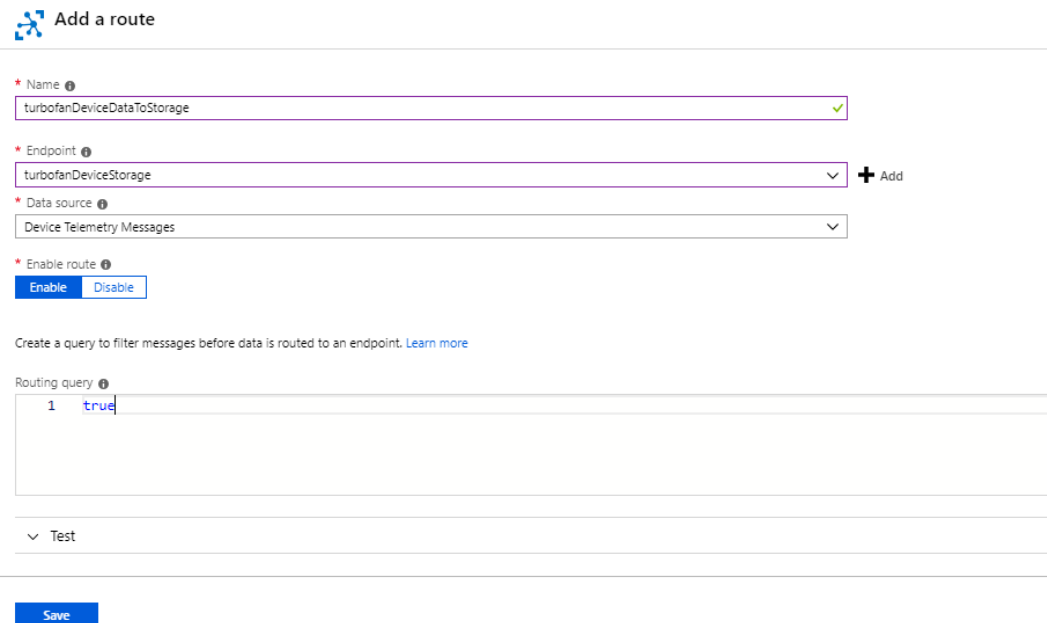
* Blob file name format ⓘ
{iothub}/{YYYY}/{MM}/{DD}/{HH}/{mm}/{partition}

The format must contain {iothub}, {partition}, {YYYY}, {MM}, {DD}, {HH} and {mm} in any order.

If multiple files are created within the same minute, the filename format would be IoEdgeMLWalkthrough/2019/01/11/17/31/0-01.

10. Back in “Add a route”, add the word “true” into the “Routing query”. The query “true” matches every message causing all incoming messages to be routed to the storage container.

11. Click 'Save'.



Add a route

* Name ⓘ
turbofanDeviceDataToStorage ✓

* Endpoint ⓘ
turbofanDeviceStorage ✓ + Add

* Data source ⓘ
Device Telemetry Messages ✓

* Enable route ⓘ
[Enable](#) [Disable](#)

Create a query to filter messages before data is routed to an endpoint. [Learn more](#)

Routing query ⓘ
1 true

Test

[Save](#)

4.6 Summary

In this section we created an IoT Hub and configured a route to an Azure Storage account. In the next section we will send data from a set of simulated devices through the IoT Hub into the storage account.

For more information see:

- [Azure IoT Fundamentals](#)
- [Configure message routing with IoT Hub](#)
- [Create an IoT hub using the Azure portal](#)

5 Generate device data

5.1 Introduction

As stated in the [IoT Edge for Machine Learning](#) white paper, this walk-through uses the “Turbofan engine degradation simulation data set” from <https://c3.nasa.gov/dashlink/resources/139/> to simulate data from a set of airplane engines for training and testing. The dataset is included in this repository and can be downloaded in zip format from <http://ti.arc.nasa.gov/c/6/>. From the accompanying readme.txt we know that:

- The data consists of multiple multivariate time series
- Each data set is divided into training and test subsets
- Each time series is from a different engine
- Each engine starts with different degrees of initial wear and manufacturing variation

In this section we use the training data subset of a single data set (FD003).

In reality, of course each engine would be an independent IoT device. Assuming you do not have a collection of internet-connected turbofan engines available, we will build a software stand-in for these devices.

The simulator is a C# program that uses the IoT Hub APIs to programmatically register virtual devices with IoT Hub.

We then read the data for each device from the NASA-provided data subset and send it to the IoT Hub using a simulated IoT Device. All the code for this section can be found in the DeviceHarness directory of the repository.

The DeviceHarness is a .NET core project written in C# consisting of 4 classes:

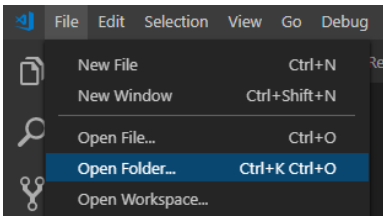
- **Program:** The entry point for execution responsible for handling user input and overall coordination.
- **TrainingFileManager:** responsible for reading and parsing the selected data file
- **CycleData:** represents a single row of data in a file converted to message format
- **TurbofanDevice:** responsible for creating an IoT Device that corresponds to a single device (time series) in the data and transmitting the data to IoT Hub via the IoT Device.

This will take about 20 minutes to complete.

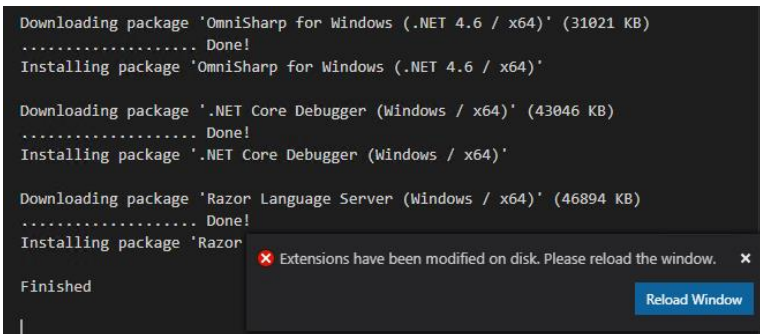
5.2 Configure Visual Studio Code and build DeviceHarness project

1. Open a remote desktop session to your VM
2. Open Visual Studio Code

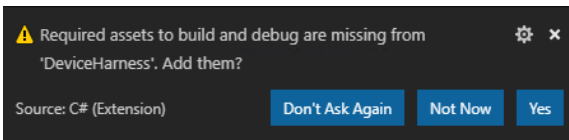
3. After Visual Studio Code finishes initializing, click on **File -> Open Folder...**



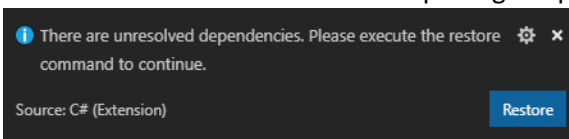
4. In the “Folder” textbox type “c:\source\IoTEdgeAndMISample\DeviceHarness” and click the “Select Folder” button.
- If OmniSharp errors appear in the output window, you’ll need to uninstall the C# extension, close/reopen VS Code, install the c# extension, and reload.
5. Since this is the first time using extensions on this machine, some extensions will update and install their dependencies. You may be prompted to update extension, if so click “Reload Window”



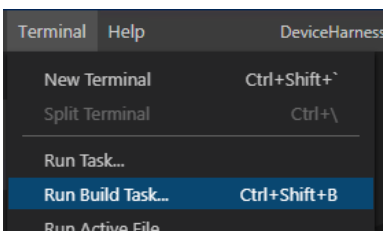
6. You will be prompted to add required assets for DeviceHarness, click “Yes” to add them
- The notification may take a few seconds to appear.
 - If you missed this notification, check the “bell” icon in the lower right-hand corner.



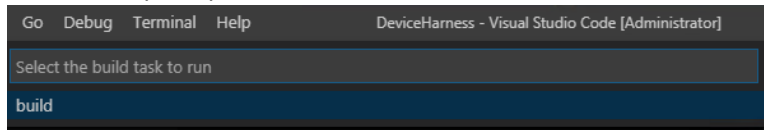
7. Then click “Restore” to restore the package dependencies



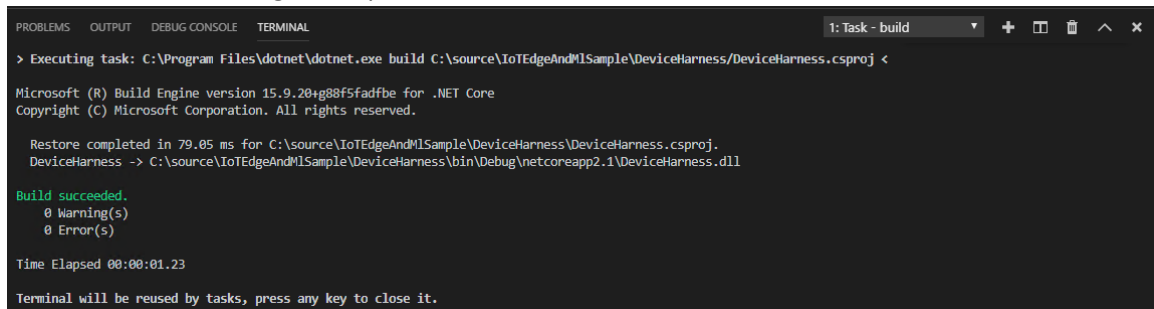
8. Validate that your environment is properly set up by triggering a build (Ctrl+Shift+B) or **Terminal->Run Build Task**



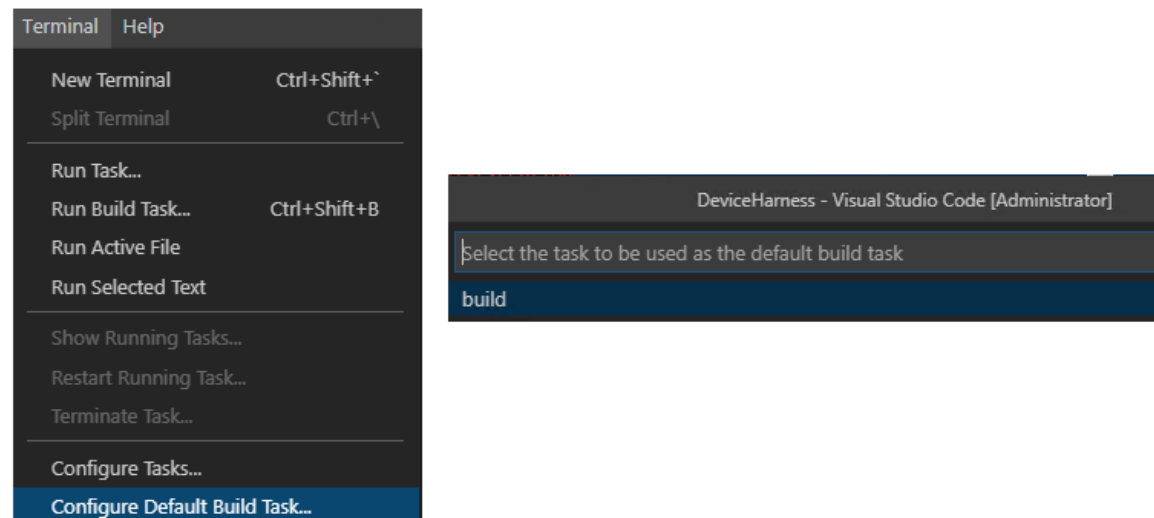
9. You will be prompted to “Select the build task to run” click on “Build”



10. The build will run and give output like:



11. You can make this build the default build task by clicking on **Terminal->Configure Default Build Task...** and choosing “build” from the “Select the task to be used as the default build task” prompt

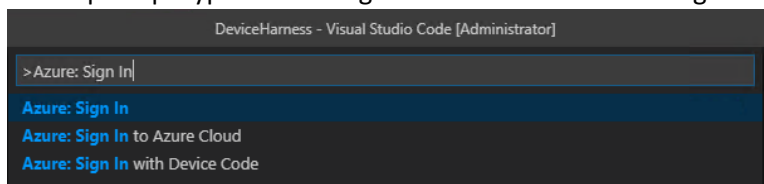


5.3 Connect to IoT Hub and run DeviceHarness

Now that we have the project building, we need to connect to the IoT Hub so that we can access the connection string and monitor the progress of the data generation.

5.3.1 Sign Visual Studio Code into Azure

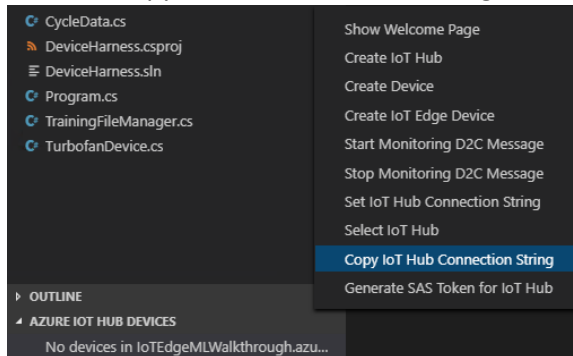
1. Sign into your Azure subscription in Visual Studio Code by opening the command palette (Ctrl+Shift+P) or **View->Command Palette...**
2. At the prompt type “Azure: Sign In” and click on “Azure:Sign In”



3. A browser window will open and prompt you for your credentials and redirect to a success page. Close the browser

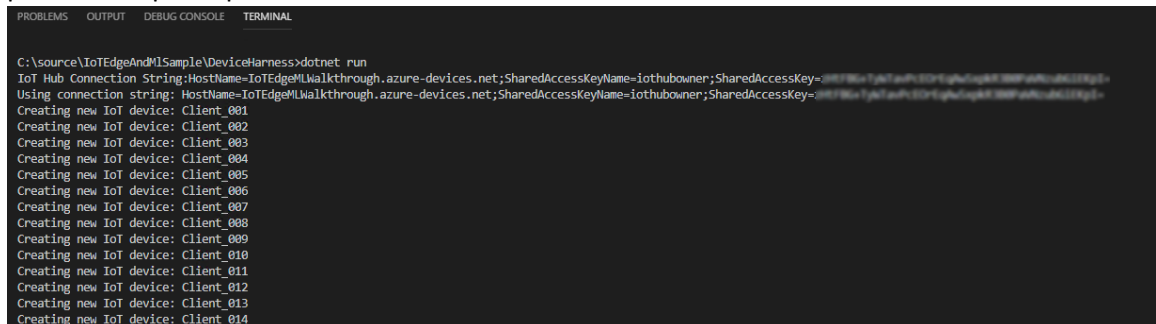
5.3.2 Connect Visual Studio Code to IoT Hub and copy hub connection string

1. In bottom section of the Visual Studio Code explorer click on the “AZURE IOT HUB DEVICES” frame to expand it
2. In the expanded frame, click on “Select IoT Hub”
3. You will be prompted to select your Azure Subscription, then your IoT Hub
4. Click on into the “AZURE IOT HUB DEVICES” frame and then click the “...” for more actions and click on “Copy IoT Hub Connection String”

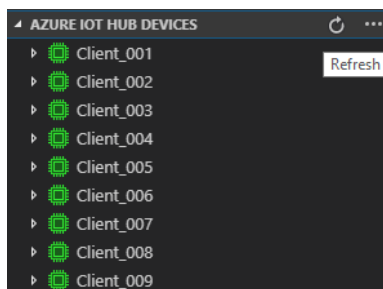


5.3.3 Run the DeviceHarness project

1. Open the Visual Studio Code terminal (Ctrl+Shift+` for new window or use an existing one)
 - a. If you do not see a prompt, hit enter.
2. Type “dotnet run”
3. When prompted for the “IoT Hub Connection String” paste the connection string copied in the previous step and press ‘Enter’



4. In the “AZURE IOT HUB DEVICE” pane click on the refresh button



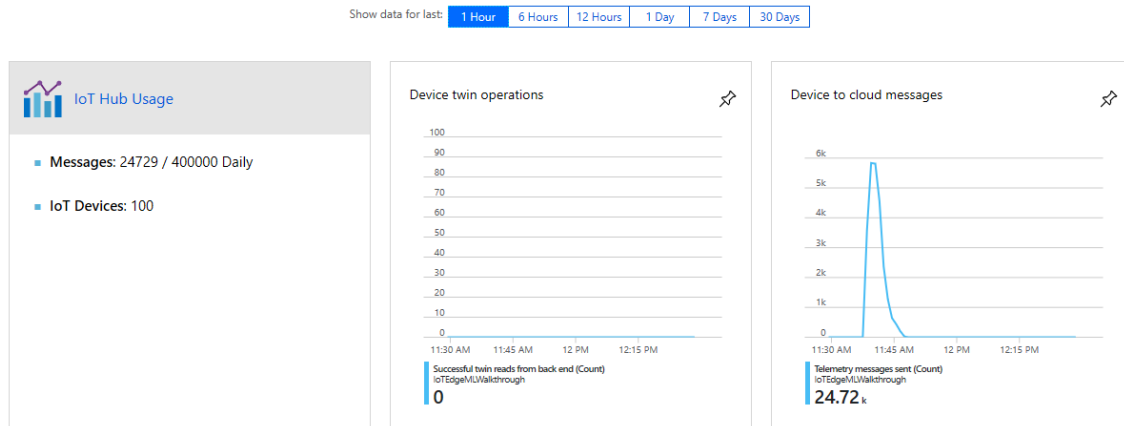
5. Note that devices are added to the IoT Hub and that the devices show up in green to indicate that data is being sent via that device.

- Let the application run to completion, which will take a few minutes.

5.4 Check IoT Hub for Activity

The data sent by the DeviceHarness went to our IoT Hub. It is easy to verify that data has reached our hub using the Azure Portal.

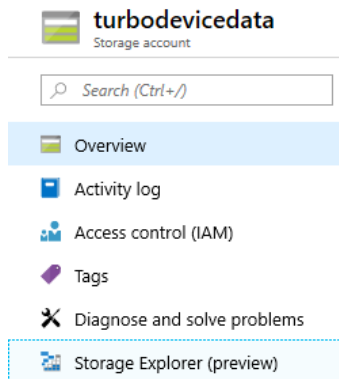
- Open <http://portal.azure.com/> and navigate to your IoT Hub
- In the overview page you should see that data has been sent to the hub:



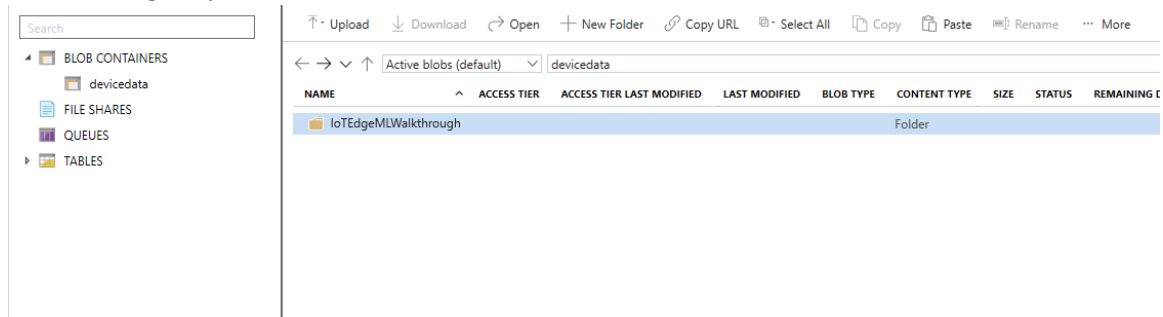
5.5 Validate data in Azure Storage

The data we just sent to the IoT Hub will have been routed to the storage container that we created above ([Create storage container](#)). Let's look at the data in our storage account.

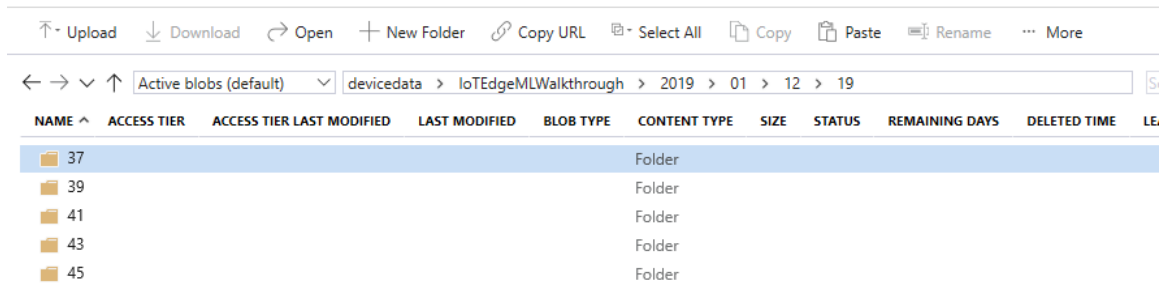
- Open <http://portal.azure.com/> and navigate to your storage account
- From the storage account navigator click on "Storage Explorer (preview)"



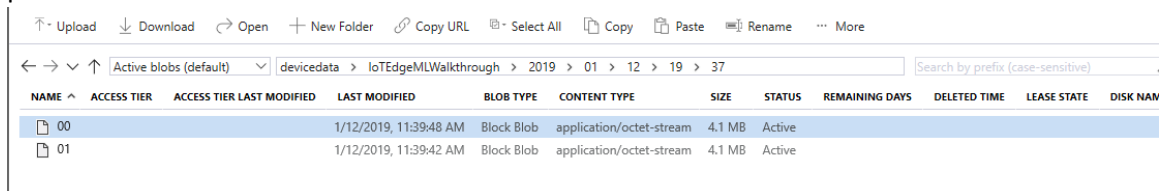
- In the storage explorer click on "BLOB CONTAINERS" then "devicedata"



- In the content pane click on the folder for the name of the IoT Hub, then the year, then the month, then the day, the hour. You will see several folders representing the minutes when the data was written.



- Click into one of those folders to find data files labelled “00” and “01” corresponding to the partition.



- The files are written in Avro (<http://avro.apache.org/>) format but double-clicking on one of these files will open another browser tab and partially render the data. If instead you are prompted to open the file in a program, you can choose VS Code and it will render fine. The result will look like:

```
Obj{}avro.codec{}null{}avro.schema{}{"type":"record","name":"Message","namespace":"Microsoft.Azure.Devices","fields":[{"name":"EnqueuedTimeUtc","type":"string"}, {"name":"Properties","type":{"type":"map","values":"string"}}, {"name":"SystemProperties","type":{"type":"map","values":"string"}}, {"name":"Body","type":{"type":"bytes"}}]}
Y%œO>fûÄiü-ê'A'¼DëÇ;82019-01-12T19:37:19.846000Z
$connectionDeviceId{}Client_007(connectionAuthMethod{}{"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}
8connectionDeviceGenerationId$636829170795318018{}content{}application/json{}contentEncoding{}utf-8{}enqueuedTime82019-01-
12T19:37:19.846000Z{}{"CycleTime":1,"OperationalSetting1":-0.0024,"OperationalSetting2":0.0001,"OperationalSetting3":100.0,"Sensor1":518.67,"Sensor2":642.05,"Sensor3":1581.59,
82019-01-12T19:37:19.862000Z{}$connectionDeviceId{}Client_001(connectionAuthMethod{}{"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}
8connectionDeviceGenerationId$636829170780502786{}content{}application/json{}contentEncoding{}utf-8{}enqueuedTime82019-01-
12T19:37:19.862000Z{}{"CycleTime":1,"OperationalSetting1":-0.0005,"OperationalSetting2":0.0004,"OperationalSetting3":100.0,"Sensor1":518.67,"Sensor2":642.36,"Sensor3":1583.23,
82019-01-12T19:37:19.862000Z{}$connectionDeviceId{}Client_004(connectionAuthMethod{}{"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}
8connectionDeviceGenerationId$636829170790971945{}content{}application/json{}contentEncoding{}utf-8{}enqueuedTime82019-01-
12T19:37:19.862000Z{}{"CycleTime":1,"OperationalSetting1":0.0019,"OperationalSetting2":-0.0004,"OperationalSetting3":100.0,"Sensor1":518.67,"Sensor2":642.03,"Sensor3":1587.05,
82019-01-12T19:37:19.862000Z{}$connectionDeviceId{}Client_006(connectionAuthMethod{}{"scope":"device","type":"sas","issuer":"iothub","acceptingIpFilterRule":null}
```

- There is no need to try to read or interpret the data right now; we will do it in subsequent steps.

5.6 Summary

In this section we used a .NET Core project to create a set of virtual devices and send data through those devices through our IoT Hub and into an Azure Storage container. This simulates a real-world scenario where physical devices send data including sensor readings, operational settings, failure signals and modes, etc. to an IoT Hub and onward into a curated storage. Once enough data has been collected, we use it to train models that predict the remaining useful life (RUL) for the device, which we will demonstrate in the next section.

6 Train and deploy ML model

6.1 Introduction

In this section, we use Azure Notebooks first to train a machine learning model using Azure Machine Learning and then to package the model as a container image that can be deployed as an Azure IoT Edge Module. The Azure Notebooks takes advantage of an Azure Machine Learning service workspace, which is a foundational block used to experiment, train, and deploy machine learning models.

The activities in this section are broken up across two notebooks.

- **01-turbofan_regression.ipynb:** This notebook walks through the steps to train and publish an ML model using Azure ML. Broadly, the steps involved are:
 1. Download, prepare, and explore the training data
 2. Use the service workspace to create and run an ML experiment
 3. Evaluate the model results from the experiment
 4. Publish the best model to the service workspace
- **02-turbofan_deploy_model.ipynb:** This notebook takes the model created in the previous notebook and uses it to create a container image ready to be deployed to an Azure IoT Edge.
 1. Create a scoring script for the model
 2. Create and publish the image
 3. Deploy the image as a web service on Azure Container Instance
 4. Use the web service to validate the model and the image work as expected

6.2 Setup Azure Notebooks

We use Azure Notebooks to host the two Jupyter Notebooks and supporting files. Here we create and configure an Azure Notebooks project. If you have not used Jupyter and/or Azure Notebooks here are a couple of introductory documents:

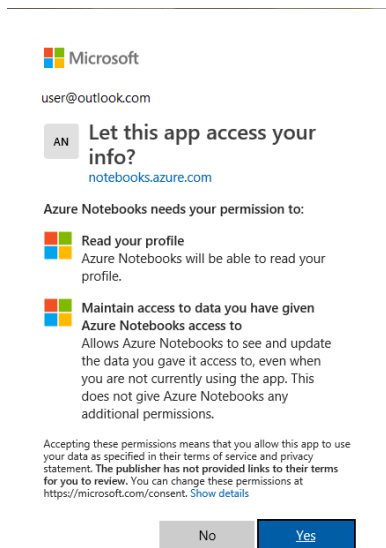
- **Quickstart:** [Create and share a notebook](#)
- **Tutorial:** [Create and run a Jupyter notebook with Python](#)

As with the Dev VM before, we choose to use Azure notebooks to ensure a consistent environment for the exercise.

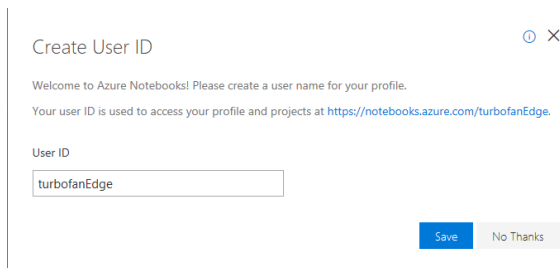
6.2.1 Create an Azure Notebooks account

Azure Notebook accounts are independent from Azure subscriptions. To use Azure Notebooks, you need to create an account.

1. Navigate to <http://notebooks.azure.com>
2. Click “Sign In” in the upper, right-hand corner of the page
3. Enter credentials using either an Azure Active Directory (AAD) or Microsoft Account
4. Grant access for the Azure Notebooks app:

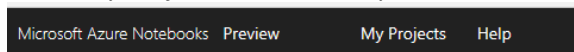


5. Create a user id for Azure Notebooks:

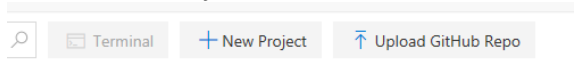


6.2.2 Create a project

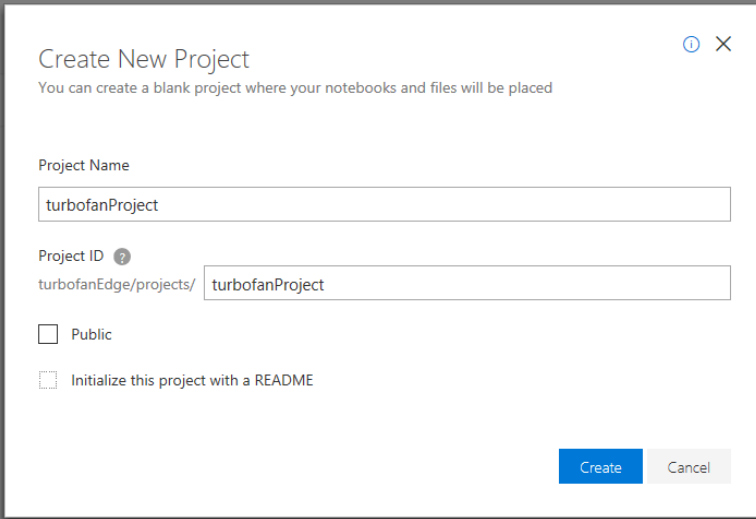
1. Click "My Projects" from the top menu bar



2. Click "+ New Project"



3. Give the project a name and an ID. There is no need for the project to be public or have a readme.



Create New Project

You can create a blank project where your notebooks and files will be placed

Project Name

turbofanProject

Project ID ?

turbofanEdge/projects/ turbofanProject

☐ Public

☐ Initialize this project with a README

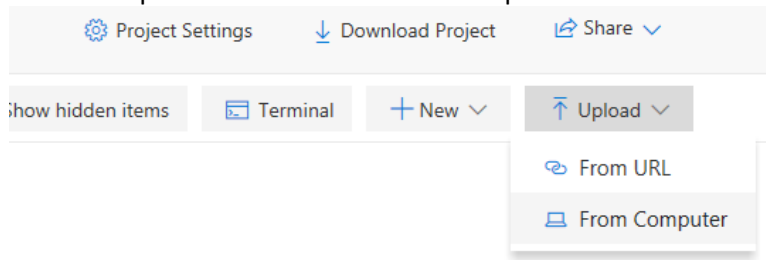
Create Cancel

6.2.3 Upload Jupyter notebooks files

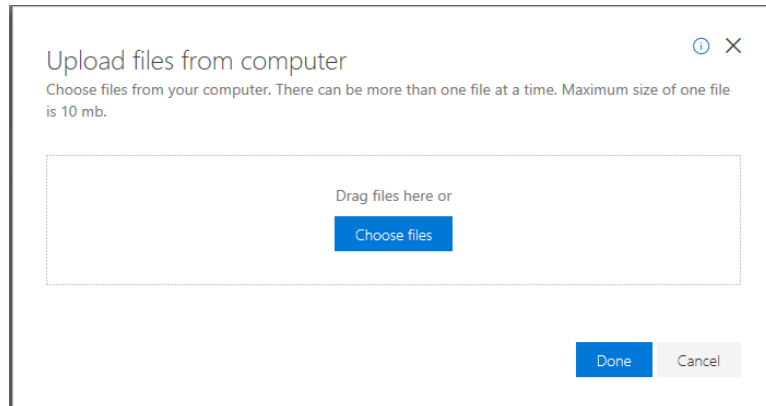
In this step, we upload files to the Azure Notebooks project we created in the last section. Specifically, the files that we upload are:

- **01-turbofan_regression.ipynb:** Jupyter notebook file that walks through the process of downloading the device harness generated data from the Azure storage account; exploring and preparing the data for training the classifier; training the model; testing the data using the test dataset found in the Test_FD003.txt file; and, finally saving the classifier model in the Machine Learning service workspace.
- **02-turbofan_deploy_model.ipynb:** Jupyter notebook that guides you through the process of using the classifier model saved in the Machine Learning service workspace to produce a container image. Once the image is created the notebook walks you through the process of deploying the image as a web service so that you can validate it is working as expected. This validated image will be deployed to our edge device in the [Create IoT Edge Modules](#) section.
- **Test_FD003.txt:** This file contains the data we will use as our test set when validating our trained classifier. We chose to use the test data as provided for the original contest as our test set for simplicity of the example.
- **RUL_FD003.txt:** This file contains the RUL for the last cycle of each device in the Test_FD003.txt file. See the **readme.txt** and the **Damage Propagation Modeling.pdf** files in the C:\source\IoTEdgeAndMISample\data\Turbofan for a detailed explanation of the data.
- **Utils.py:** Contains a set of Python utility functions for working with data. The first notebook contains a detailed explanation of the functions.
- **README.md:** Readme describing the use of the notebooks.

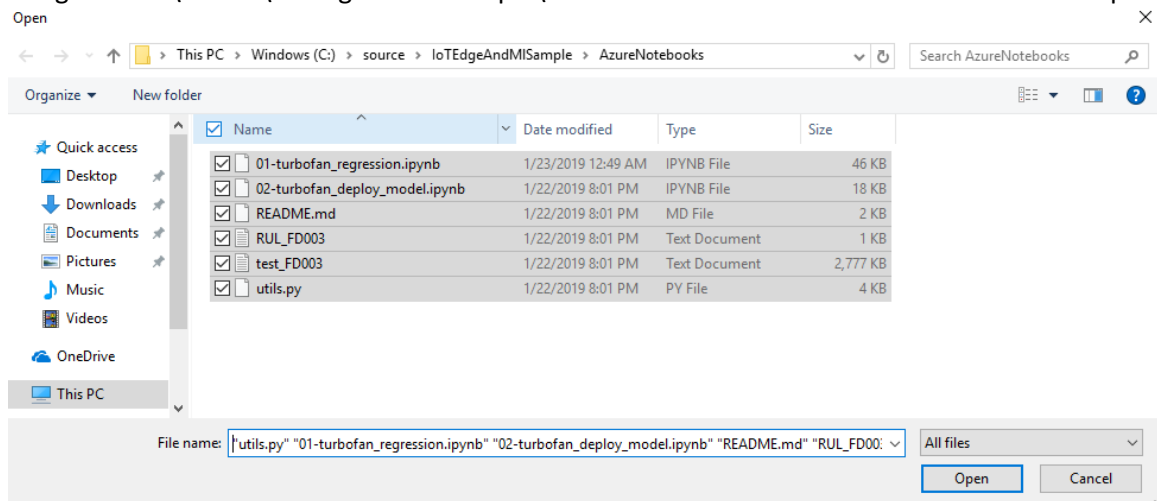
1. Click on “Upload” and choose “From Computer”



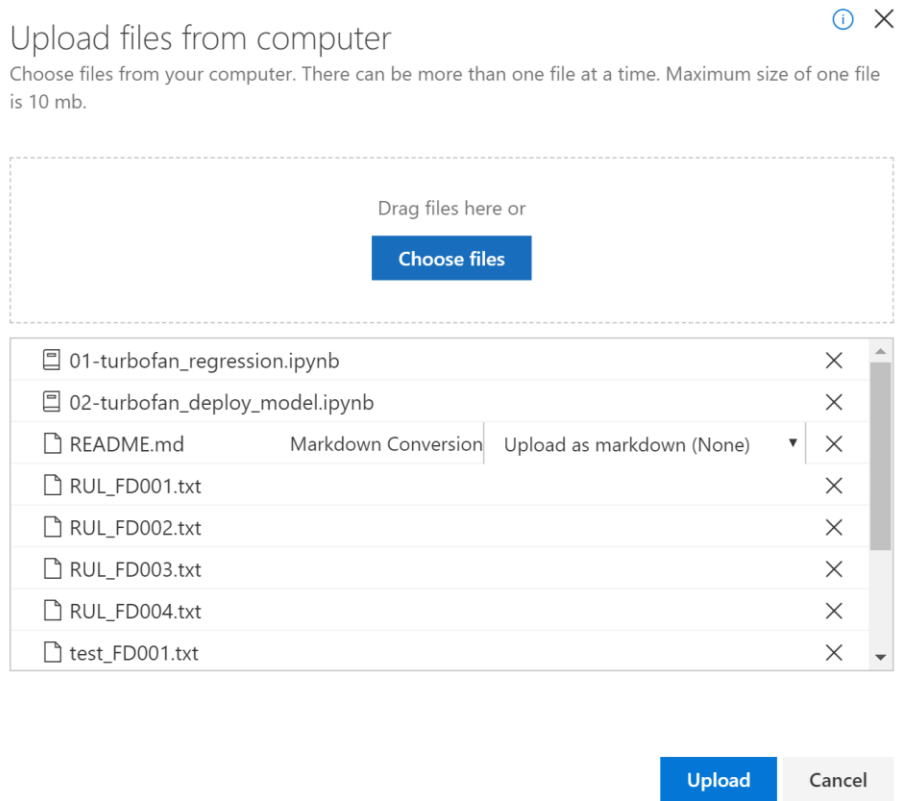
2. Click on “Choose files”



3. Navigate to C:\source\IoTEdgeAndMISample\AzureNotebooks and select all files and click “Open”



4. Click on upload

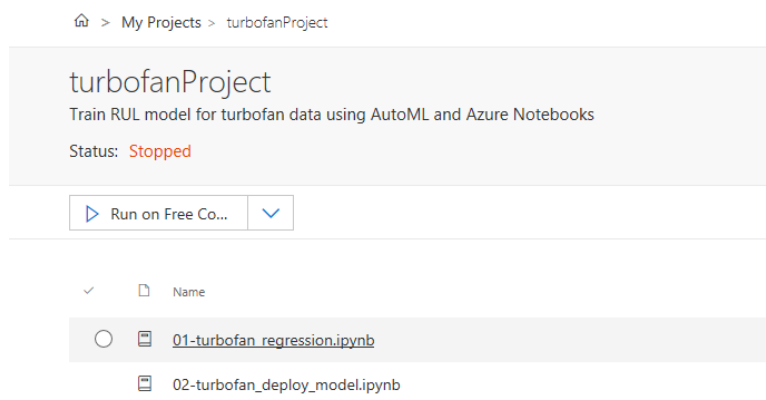


5. Click "Done"

6.3 Run Azure Notebooks

Now that that project is created, run the 01-turbofan_regression.ipynb by:

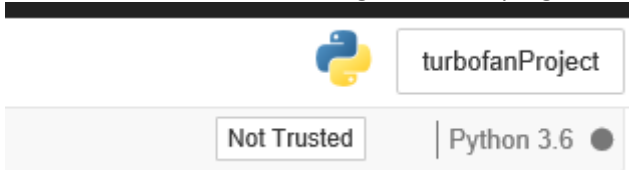
1. From the turbofan project page click on "01-turbofan_regression.ipynb"



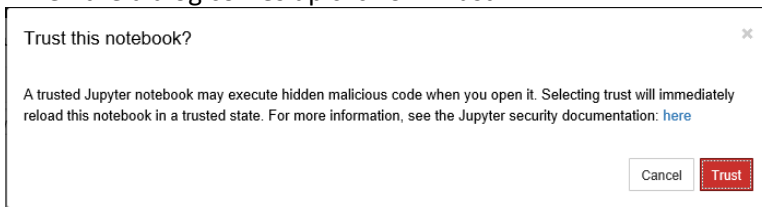
2. Choose the Python 3.6 Kernel from the dialog and click “Set Kernel”:



3. Click on the 'Not Trusted' widget in the top right of the notebook:



4. When the dialog comes up click on 'Trust'



5. Follow the instructions in the notebook.

- **Helpful shortcuts:**
 - Ctrl + Enter runs a cell
 - Shift + Enter runs a cell and navigates to the next cell
- **Note:** when a section is being run, it will have an asterisk between the square brackets (e.g. [*]). When it is complete, the asterisk will be replaced with a number and relevant output may appear below. Sections usually build off the previous ones so wait to run the next section after the previous one has completed.

When you have finished running the 01-turbofan_regression.ipynb return to the project page and click on “02-turbofan_deploy_model.ipynb” to run the second notebook.

6.4 Summary

In this section, we used 2 Jupyter Notebooks running in Azure Notebooks to use the data from the turbofan devices to train a remaining useful life (RUL) classifier, to save the classifier as a model, to create a container image, and to deploy and test the image as a web service.

7 Configure IoT Edge device

7.1 Introduction

In this section we will configure an Azure Virtual Machine running Linux as an Azure IoT Edge device. We will set up the edge device to act as a transparent gateway. The transparent gateway configuration allows devices to connect to that Azure IoT Hub through the gateway without being aware that the gateway exists. At the same time, a user interacting with the devices in IoT Hub is unaware of the intermediate gateway device. Ultimately, we will use the transparent gateway to add edge analytics to our system by adding edge modules to the gateway.

7.2 Generate certificates

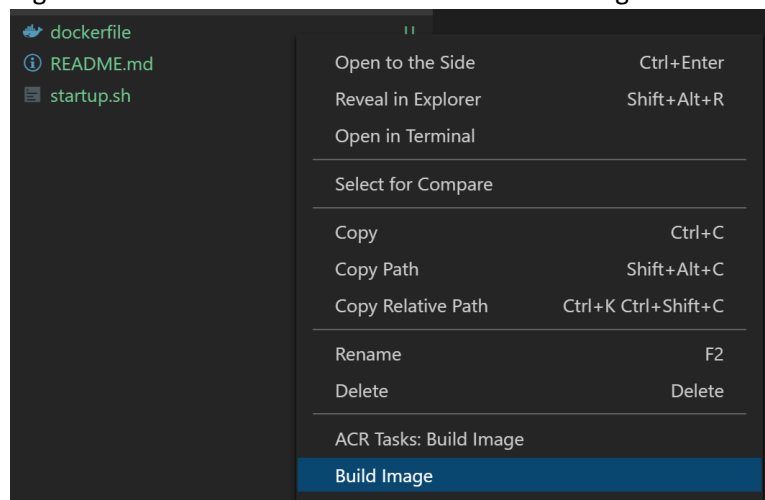
For a device to function as a gateway it needs to be able to securely connect to downstream devices. Azure IoT Edge allows you to use a public key infrastructure (PKI) to set up secure connections between devices. In this case, we're allowing a downstream device to connect to an IoT Edge device acting as a transparent gateway. To maintain reasonable security, the downstream device should confirm the identity of the Edge device. A detailed explanation of certificate use in IoT Edge can be found in this [document](#).

In this section, we create the self-signed certificates using a Docker image that we will build and run.

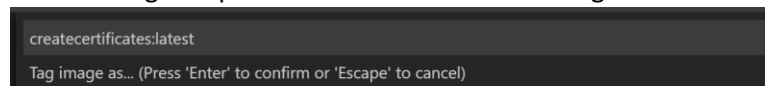
1. Login to your development machine
2. Create a directory on the VM. Open a command line and run the command

```
mkdir c:\edgecertificates
```

3. Start "Docker for Windows" from the Windows Start menu
4. Open Visual Studio Code
5. **File->Open Folder...** and choose "C:\source\IoTEdgeAndMISample\CreateCertificates"
6. Right-click on the dockerfile and choose "Build Image"



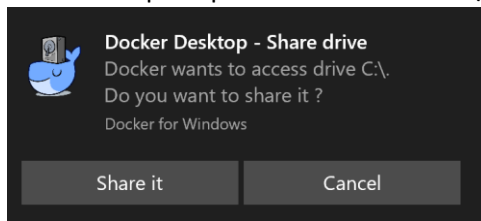
7. In the dialog accept the default value for the image name and tag, "createcertificates:latest"



8. Wait for the build to complete
9. In the Visual Studio Code terminal window run the command:

```
docker run --name createcertificates --rm -v  
c:\edgeCertificates:/edgeCertificates createcertificates  
/edgeCertificates
```

10. Docker will prompt for access to the C:\ drive, click “Share it”



11. Provide your credentials when prompted
12. Once the container finishes running check for the files in c:\edgecertificates:

```
C:\edgeCertificates\certs\azure-iot-test-only.root.ca.cert.pem
C:\edgeCertificates\certs\new-edge-device-full-chain.cert.pem
C:\edgeCertificates\certs\new-edge-device.cert.pem
C:\edgeCertificates\certs\new-edge-device.cert.pfx
C:\edgeCertificates\private\new-edge-device.key.pem
```

7.3 Upload certificates to Azure Key Vault

To store our certificates securely and to make them accessible from multiple devices we will upload the certificates into Azure Key Vault. As you can see from the list above, we have 2 types of certificate files. We will treat the PFX¹ as Key Vault Certificates to be uploaded to Key Vault. The PEM² files are plain text and we will treat them as Key Vault Secrets. We will use the Key Vault associated with the Azure Machine Learning service workspace we created by running the [Azure Notebooks](#).

1. Navigate to the Azure Portal (<http://portal.azure.com>) and find your Machine Learning service workspace
2. From the overview page of the Machine Learning service workspace copy the name of the Key Vault



3. On your development machine, open PowerShell run the command

```
C:\source\IoTEdgeAndMlSample\CreateCertificates\upload-keyvaultcerts.ps1
-SubscriptionId <subscriptionId> -KeyVaultName <keyvaultname>
```

4. Login to azure if you are prompted.
5. The script will run for few minutes with output like

```
Uploading azure-iot-test-only.root.ca.cert.pem...
Uploading new-edge-device-full-chain.cert.pem...
Uploading new-edge-device.cert.pem...
Uploading new-edge-device.cert.pfx...
Uploading new-edge-device.key.pem...
Results:
Certificate                                KeyVaultName                                KeyVaultId
-----
azure-iot-test-only.root.ca.cert.pem      https://turbofankeyvaultcojeflo.vault.azure.net/443/secrets/azure-iot-test-only-root-ca-cert-pem/38c64784ed824153b5aa5e5e4421b3a7
new-edge-device-full-chain.cert.pem      https://turbofankeyvaultcojeflo.vault.azure.net/443/secrets/new-edge-device-full-chain-cert-pem/24ae146d3c6e4c86b205d7609eeed475
new-edge-device.cert.pem                  https://turbofankeyvaultcojeflo.vault.azure.net/443/secrets/new-edge-device-cert-pem/5aeef731b8be495d8413cbbab9985dec
new-edge-device.cert.pfx                  https://turbofankeyvaultcojeflo.vault.azure.net/443/certificates/new-edge-device-cert-pfx/52c0b5478055480287aa0e9a8bf82273
new-edge-device.key.pem                   https://turbofankeyvaultcojeflo.vault.azure.net/443/secrets/new-edge-device-key-pem/ad2fe078354183b0b1fa088f78ed1
```

¹ Not necessary for completing the walk-through, but more information about PFX files can be found at: https://en.wikipedia.org/wiki/PKCS_12

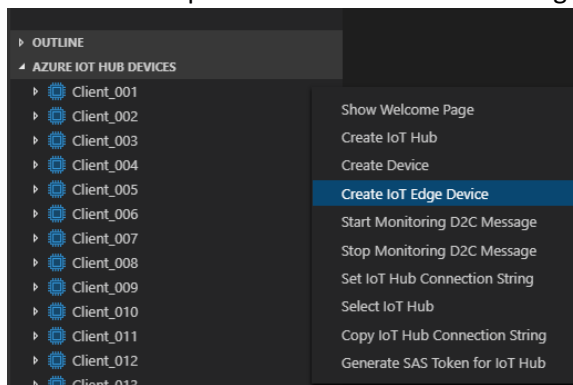
² More information on PEM can be found in: https://en.wikipedia.org/wiki/Privacy-Enhanced_Mail

7.4 Create Azure IoT Hub Edge device

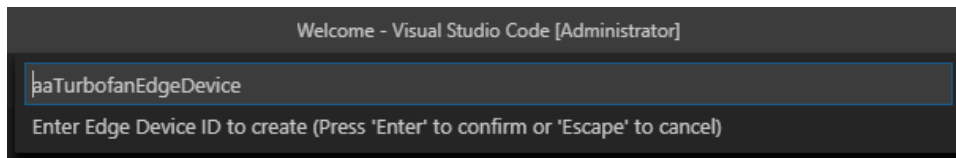
To connect an Azure IoT Edge device to an IoT Hub, we first create a representation of the edge device in the hub. We take the connection string from the hub device representation and use it to configure the runtime on our edge device. Once the edge device has been configured and connects to the hub, we are able to deploy modules and send messages. We can also change the configuration of the edge device by changing the configuration of the corresponding device representation in IoT hub.

You can create an Azure IoT Edge device in the hub in multiple ways including creating using the [Azure Portal](#), using [Azure CLI](#), or, as we do here using Visual Studio Code.

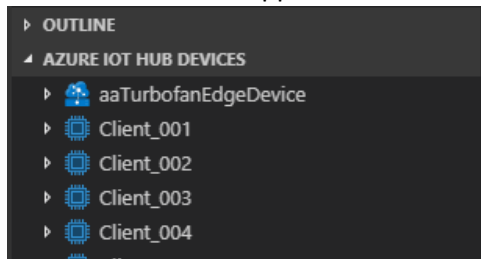
1. On your development machine open Visual Studio Code
2. Open the “AZURE IOT HUB DEVICES” panel from the Visual Studio Code explorer view
3. Click on the ellipsis and choose “Create IoT Edge Device”



4. Give the device a name. For convenience, we use “aaTurbofanEdgeDevice” so it sorts ahead of all of the Client_nnn devices we created earlier through the device harness to send the test data.



5. The new device will appear in the devices view



7.5 Deploy Azure VM

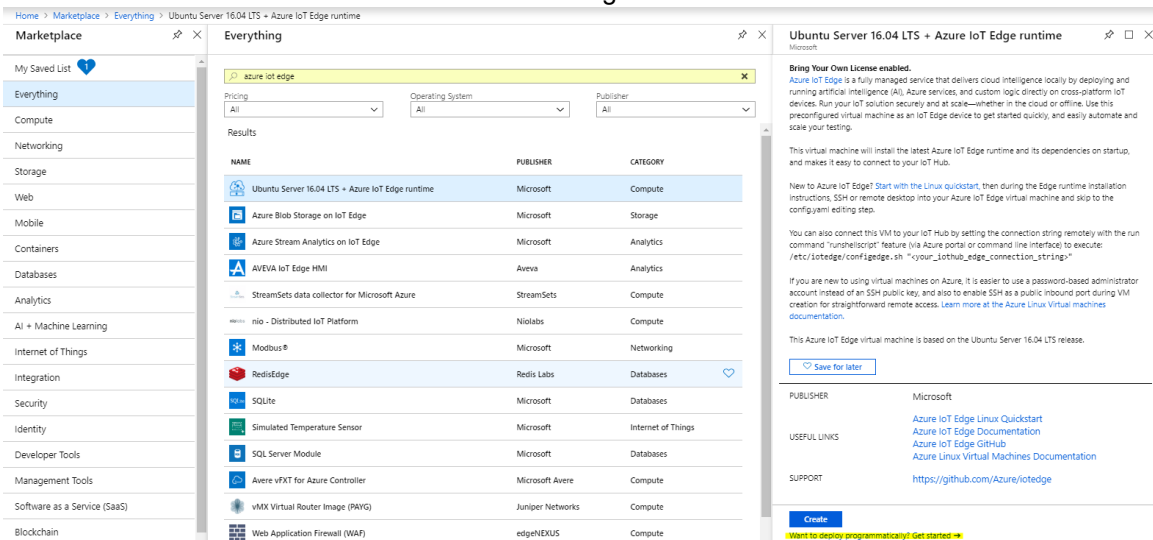
We use the [Azure IoT Edge on Ubuntu](#) image from the Azure Marketplace to create our edge device for this walk-through. The *Azure IoT Edge on Ubuntu* image installs the latest Azure IoT Edge runtime and its

dependencies on startup³. We will deploy the VM using a PowerShell script, Create-EdgeVM.ps1; an ARM template, IoTEdgeVMTemplate.json; and a shell script, install packages.sh.

7.5.1 Enable programmatic deployment

To use the image from the marketplace in a scripted deployment we need to enable programmatic deployment for the image. To do that:

1. Login to <http://portal.azure.com>
2. From the left navigator click on “All services”
3. Type “marketplace” into the filter and then click on “Marketplace”
4. Type “Azure IoT Edge” into the “Search Everything” box and hit enter
5. Click on “Ubuntu Server 16.04 LTS + Azure IoT Edge runtime”



6. Click the hyperlink “Want to deploy programmatically? Get started ->” below the “Create” button

³ See [Install the Azure IoT Edge runtime on Linux \(x64\)](#) for detailed installation steps

- Click on the “Enable” button in the “Configure Programmatic Deployment” page then click “Save”

Configure Programmatic Deployment

Use API calls, ARM templates, or the PowerShell console to automatically deploy without using the Azure portal. You'll only need to do this once—the settings you choose will be used each time you deploy.

Offer details

Ubuntu Server 16.04 LTS + Azure IoT Edge runtime
by Microsoft
[Terms of use](#) | [privacy policy](#)

Pricing does not include [Azure infrastructure costs](#) (e.g., virtual machine compute time or storage) and is based on the pricing tier you select at the time of deployment. The pricing above applies only to Azure subscriptions purchased from Microsoft. For Azure subscriptions purchased from a reseller, contact your reseller for pricing. Neither subscription credits nor monetary commitment funds may be used to purchase non-Microsoft offerings. These purchases are billed separately. If any Microsoft products are included in the above offering(s) (e.g., Windows Server or SQL Server), such products are licensed by Microsoft and not by any third party.

Terms of use

By enabling programmatic purchases for the subscriptions selected below, I (a) agree to the legal terms and privacy statement(s) associated with each offering above, (b) for Azure subscriptions purchased from Microsoft, authorize Microsoft to charge or bill my current payment method for the fees associated with my use of the offering(s), including applicable taxes, with the same billing frequency as my Azure subscription, until I discontinue use of the offering(s), and (c) agree that Microsoft may share my contact information, and transaction details associated with my purchase of the above offering(s), with any third-party vendors, if listed above. Microsoft does not provide rights for third-party products or services. See the [Azure Marketplace Terms](#) for additional terms.

Choose the subscriptions

Select the Azure subscriptions for which you would like to enable programmatic deployments of the above offering(s)

SUBSCRIPTION NAME	SUBSCRIPTION ID	STATUS
107-000-000-000	00000000-0000-0000-0000-000000000000	<div>Enable</div>

Save

Discard

- You will see a success notification

Configure Programmatic Deployment

Configuration updates completed.

7.5.2 Create virtual machine

Next run the script to create the edge device virtual machine.

- Open a PowerShell window and navigate to C:\source\IoTEdgeAndMISample\EdgeVM”
- Type the command:

```
.\Create-EdgeVm.ps1
```

- When prompted provide values for each parameter. For subscription, resource group and location we recommend you use the same as you have for all resources throughout the walk-through
 - Azure Subscription ID: found in the Azure Portal
 - Resource Group Name: memorable name for grouping the resources for your walk-through
 - Location: Azure location where the virtual machine will be created (e.g. West US 2, North Europe, see full [list](#))
 - AdminUsername: the name for the admin account you want to create and use on the virtual machine
 - AdminPassword: the password to set for the AdminUsername on the VM

4. For the script to be able to set up the VM, you need to login to Azure with the credentials associated with the Azure Subscription you are using
5. The script confirms the information for the creation of your VM. Press 'y' or 'Enter' to continue
6. The script will run for several minutes as it executes the steps:
 - Create the Resource Group if it does not exist
 - Deploy the virtual machine
 - Install the Azure CLI ([details](#))
7. Note the script outputs the SSH connection string for connecting to the VM, copy this for the next step

```
The VM is ready.
Visit the Azure Portal (http://portal.azure.com).
- Virtual machine name: IoTEdge-7s3usse
- Resource group: iotedge
- Subscription:
- Connect with: ssh -l markmyd iotedge-7s3usse.westus2.cloudapp.azure.com
```

7.6 Connect to the VM using SSH

The next several steps will be configuring the Azure VM we just created. The first step is to get connected to the virtual machine.

1. Open a command shell and paste the ssh connection string you copied from the script output

```
ssh -l <username> iotedge-<suffix>.<region>.cloudapp.azure.com
```

2. You will be prompted to validate the authenticity of the host, type "yes" and hit enter.

```
C:\Users\markmyd>ssh markmyd@iotedge-enpqdkm.westus2.cloudapp.azure.com
The authenticity of host 'iotedge-enpqdkm.westus2.cloudapp.azure.com (52.191.129.230)' can't be established.
ECDSA key fingerprint is SHA256:2tdL90KfH/p1eaUZIRMCv12T1Tl+FFhQLGrN1bwQ1iQ.
Are you sure you want to continue connecting (yes/no)?
```

3. When prompted provide your password
4. Ubuntu will display a welcome message and then you should see a prompt like
 <username>@<machinename>:~\$

```
@IoTEdge-byyy63o:~$
```

7.7 Download Key Vault certificates

We uploaded the certificates to Key Vault to make them available for our edge device and our leaf device. In this step, we download the certificates to the edge device.

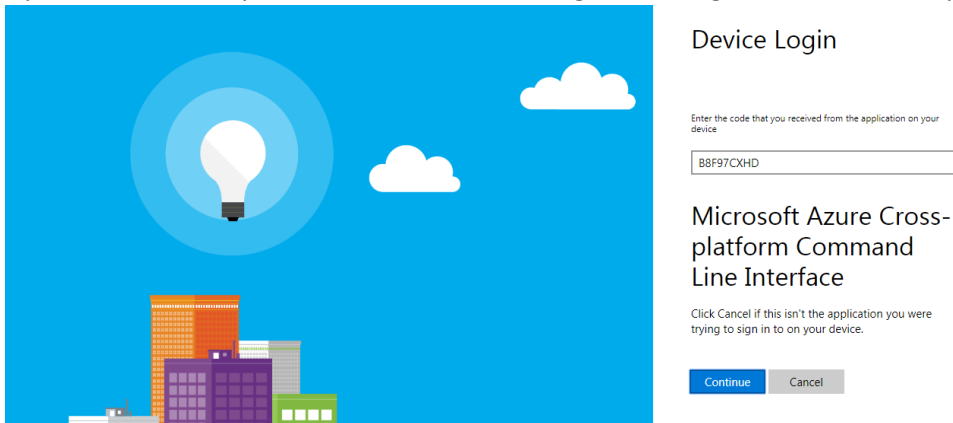
1. From the ssh session on the Linux VM type:

```
az login
```

2. You will be prompted to open a browser to <https://microsoft.com/devicelogin> and provide a unique code

```
@IoTEdge-enpqdkm:~$ az login
To sign in, use a web browser to open the page https://microsoft.com/devicelogin and enter the code B8F97CXHD to authenticate.
```

3. Open a browser on your local machine and navigate to the given location and type in the code



CM

4. Hit continue and proceed through the login process for Azure
5. When you complete the login, the remote VM will login and list your Azure subscriptions
6. At the ssh prompt type

```
az account set --subscription <subscription id>
```

7. Create a directory on the VM for the certificates

```
sudo mkdir /edgeMlCertificates
```

8. Download new-edge-device-full-chain.cert.pem, new-edge-device.key.pem and azure-iot-test-only.root.ca.cert.pem

```
key_vault_name="<key vault name>"

sudo az keyvault secret download --vault-name $key_vault_name --name
new-edge-device-full-chain-cert-pem -f /edgeMlCertificates/new-edge-
device-full-chain.cert.pem
sudo az keyvault secret download --vault-name $key_vault_name --name
new-edge-device-key-pem -f /edgeMlCertificates/new-edge-device.key.pem
sudo az keyvault secret download --vault-name $key_vault_name --name
azure-iot-test-only-root-ca-cert-pem -f /edgeMlCertificates/azure-iot-
test-only.root.ca.cert.pem
```

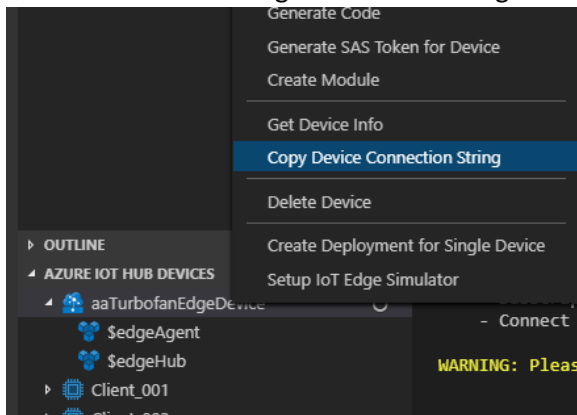
7.8 Update the edge device configuration

The Edge runtime uses the file `/etc/iotedge/config.yaml` to persist its configuration. We need to update 3 pieces of information in this file:

1. **Device connection string:** the device connection string for this device needs to be set to the value of the connection string for the device we created in the hub
2. **Certificates:** the edge runtime needs to be told which certificates it should use for connections made with downstream devices
3. **Hostname:** the hostname needs to exactly match the fully qualified domain name (FQDN) of the VM edge device.

The *Ubuntu Server 16.04 LTS + Azure IoT Edge runtime* image that we used to create the Edge VM comes with a shell script that updates the `config.yaml` with the connection string.

1. In Visual Studio Code right-click on the edge device then click “Copy Device Connection String”



2. In your ssh session run the command

```
sudo /etc/iotedge/configedge.sh  
"<your_iot_hub_edge_device_connection_string>"
```

Next we will update the certificates and hostname by directly editing the config.yaml.

1. Open config.yaml with the command

```
sudo nano /etc/iotedge/config.yaml
```

2. Update the certificates section of the config.yaml by removing the leading # and setting the path so this (right-click in the Linux terminal to paste copied text):

```
# certificates:  
#   device_ca_cert: "<ADD PATH TO DEVICE CA CERTIFICATE HERE>"  
#   device_ca_pk: "<ADD PATH TO DEVICE CA PRIVATE KEY HERE>"  
#   trusted_ca_certs: "<ADD PATH TO TRUSTED CA CERTIFICATES HERE>"
```

Becomes:

```
certificates:  
  device_ca_cert: "/edgeMlCertificates/new-edge-device-full-  
chain.cert.pem"  
  device_ca_pk: "/edgeMlCertificates/new-edge-device.key.pem"  
  trusted_ca_certs: "/edgeMlCertificates/azure-iot-test-  
only.root.ca.cert.pem"
```

Note: it is important to make sure the “certificates:” has no preceding whitespace and that each of the certificates is preceded by 2 spaces.

Note: right clicking in nano will paste the contents of your clipboard to the current cursor position. So replace the string use your keyboard arrows to navigate to the string you want to replace, delete the string, then right-click to paste from the buffer.

3. Find your Edge virtual machine in the Azure Portal on the 'Overview' page copy the DNS name (FQDN of the machine)
4. Paste the FQDN into the hostname section of the config.yml. Make sure that the name is all lowercase:

```
hostname: '<machinename>.<region>.cloudapp.azure.com'
```

5. Save and close the file (Ctrl+X, Y, Enter)
6. Restart the iotedge daemon:

```
sudo systemctl restart iotedge
```

7. Check the status of the IoT Edge Daemon (after the command, type “:q” to exit)

```
systemctl status iotedge
```

8. If you see errors (colored text prefixed with “[ERROR]”) in the status Examine daemon logs for detailed error information

```
journalctl -u iotedge --no-pager --no-full
```

7.9 Disable process identification

While in preview, Azure Machine Learning does not support the process identification security feature enabled by default with IoT Edge. Below are the steps to disable it. This is, however, not suitable for use in production.

1. On your edge device vm get the ipaddress for the docker0 interface and note the inet ipaddress

```
ifconfig docker0
```

```
$ ifconfig docker0
docker0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 172.17.0.1 netmask 255.255.0.0 broadcast 172.17.255.255
    inet6 fe80::42:c1ff:fe42:732a prefixlen 64 scopeid 0x20<link>
    ether 02:42:c1:42:73:2a txqueuelen 0 (Ethernet)
    RX packets 3030 bytes 230177 (230.1 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 2271 bytes 1050082 (1.0 MB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

2. Open config.yaml for editing:

```
sudo nano /etc/iotedge/config.yaml
```

3. Update the connect section, for example if you docker0 ip is 172.17.0.1:

```
connect:
  management_uri: "http://172.17.0.1:15580"
  workload_uri: "http://172.17.0.1:15581"
```

4. Enter the same addresses in the listen section of the configuration. For example:

```
listen:
  management_uri: "http://172.17.0.1:15580"
  workload_uri: "http://172.17.0.1:15581"
```

5. Save and exit the config.yaml file (CTRL+X,Y,ENTER)
6. Create an environment variable IOTEDGE_HOST with the management_uri address

```
export IOTEDGE_HOST="http://172.17.0.1:15580"
```

7. To make it persistent open /etc/environment:

```
sudo nano /etc/environment
```

8. Add to the end of the file:

```
IOTEDGE_HOST="http://172.17.0.1:15580"
```

9. Save and exit the environment file (CTRL+X,Y,ENTER)
10. Restart the iotedge daemon:

```
sudo systemctl restart iotedge
```

11. Check the status of the IoT Edge Daemon

```
systemctl status iotedge
```

7.10 Summary

We just completed configuring an Azure VM as Azure IoT Edge Transparent Gateway. We started by generating test certificates, which we uploaded to Azure Key Vault. Next, we used a script and ARM template to deploy the VM with the "Ubuntu Server 16.04 LTS + Azure IoT Edge runtime" image from the Azure marketplace. The script took the extra step of installing the Azure CLI ([Install Azure CLI with apt](#)). With the VM up and running we connected via ssh, logged into azure, downloaded certificates from Key Vault, and made several updates to the configuration of the IoT Edge Runtime by updating the config.yaml file. For more information about using IoT Edge as a gateway see [How an IoT Edge device can be used as a gateway](#). For more details on how to configure an IoT Edge device as a transparent gateway see [Configure an IoT Edge device to act as a transparent gateway](#).

8 Create IoT Edge Modules

8.1 Introduction

Edge Hub facilitates module to module communication. Using Edge Hub as a message broker keeps modules independent from each other. Modules only need to specify the inputs on which they accept messages and the outputs to which they write messages.

We want the Edge device to accomplish four things for us:

1. Receive data from the leaf devices
2. Predict RUL for the device that sent the data
3. Send a message with only the RUL for the device to IoT Hub (this could be modified to only send data if the RUL drops below some level)
4. Persist the device data received by Edge to a local file. This data file is then periodically uploaded via file upload to the hub in order to refine training of the ML model. Using file upload instead of constant message streaming is significantly more cost effective.

To accomplish this, we will use 3 custom modules:

- **RUL Classifier:** The turboFanRulClassifier module we created in [02-turbofan_deploy_model.ipynb](#) is a standard machine learning module, which exposes an input called "amlInput" and an output called "amlOutput". The "amlInput" expects its input to look exactly like the input that we sent to the ACI based web service. Likewise, "amlOutput" returns the same data as the web service.
- **Avro writer:** This module receives messages on the "avroModuleInput" input and persists the message in Avro format to disk for later upload to IoT Hub.

- **Router Module:** The router module receives messages from downstream leaf devices, formats and sends the messages to the classifier. The module then receives the messages from the classifier and forwards the message onto the Avro writer module. Finally, the module sends just the RUL prediction to the IoT Hub.

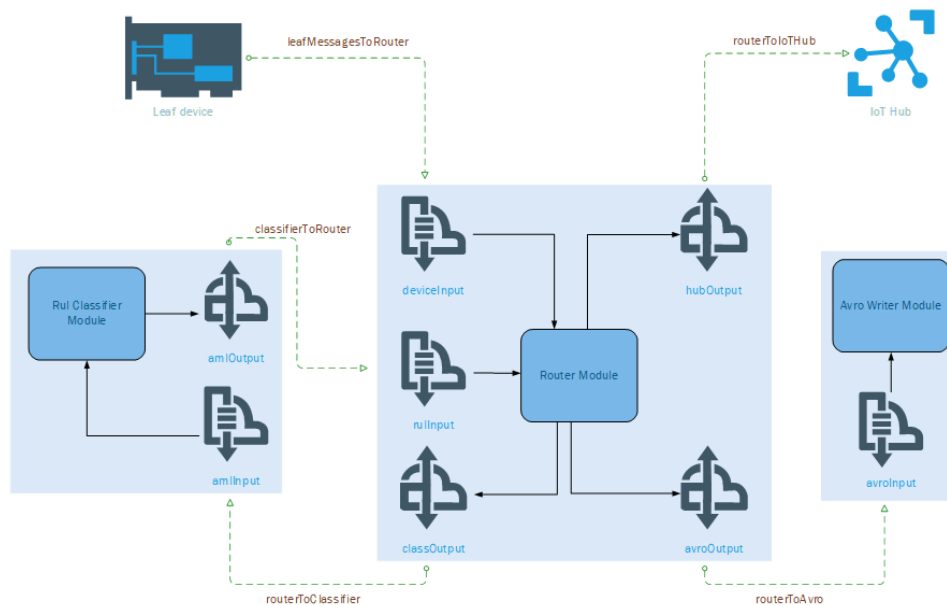
Inputs:

- **deviceInput:** receives messages from leaf devices
- **rulInput:** receives messages from the "amlOutput"

Outputs:

- **classify:** sends messages to "amlInput"
- **writeAvro:** sends messages "avroModuleInput"
- **toIoTHub:** sends messages to \$upstream, which passes the messages to the connected IoT Hub

The diagram below, shows the modules, inputs, outputs and the Edge Hub routes for the full solution:

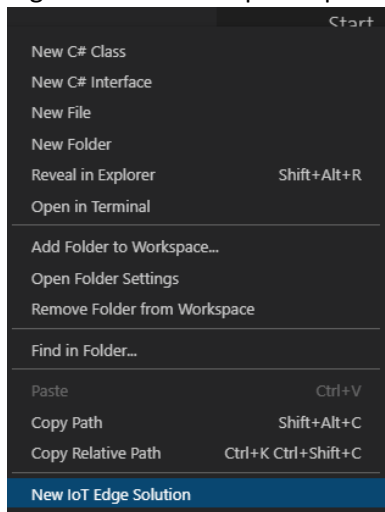


8.2 Create an Edge Solution with RUL Classifier Module

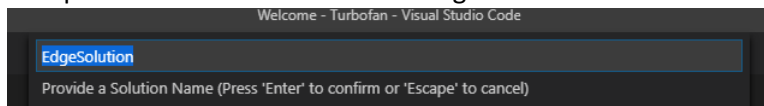
During execution of the second of our two Azure Notebooks, we created and published a container image containing our RUL model. Azure Machine Learning (AML), as part of the image creation process, built in the pieces to make the image deployable as an Azure IoT Edge module. In this step, we are going to create an Azure IoT Edge solution using the "Azure Machine Learning" module and point the module to the image we published using Azure Notebooks.

1. RDP into your development machine
2. Open folder C:\source\IoTEdgeAndMISample in Visual Studio Code

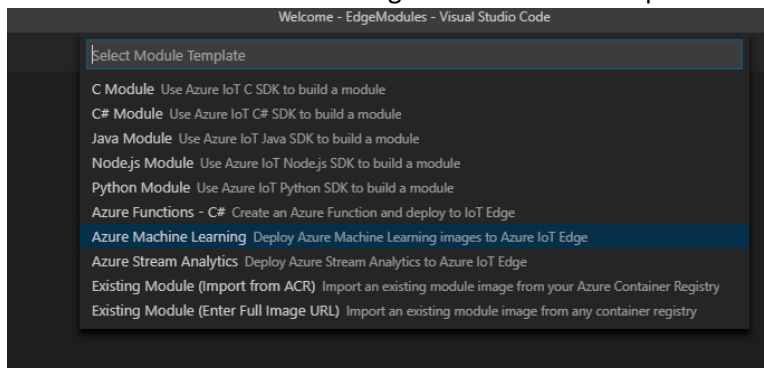
3. Right click on the explorer panel (in the blank space) and choose “New IoT Edge Solution”



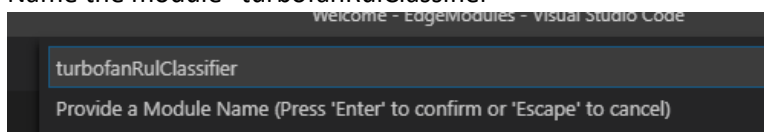
4. Accept the default solution name “EdgeSolution”



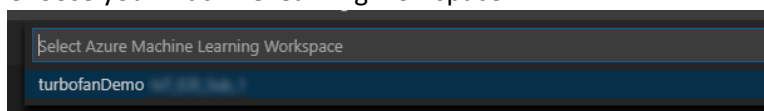
5. Choose “Azure Machine Learning” as the module template



6. Name the module "turbofanRulClassifier"



7. Choose your machine learning workspace



8. Select the image you created while running the Azure Notebook
9. Look at the solution and notice the files that have been created:

- **deployment.template.json:** This file contains the definition of each of the modules in the solution. There are three sections to pay attention to in this file:
 1. **Registry credentials:** defines the set of custom container registries you are using in your solution. Right now, it should contain the registry from your machine learning workspace, which is where your AML image was stored. You can have any number of container registries, but for simplicity we will use this one registry for all modules

```
"registryCredentials": {
  "turbofanacrptzrofnh": {
    "username": "$CONTAINER_REGISTRY_USERNAME_<your
registry>",
    "password": "$CONTAINER_REGISTRY_PASSWORD_<your
registry>",
    "address": "<your registry>.azurecr.io"
  }
}
```

2. **Modules:** this section contains the set of user defined modules that go with this solution. You will notice that this section currently contains 2 modules, tempSensor and turbofanRulClassifier. The tempSensor was installed by the Visual Studio Code template and we don't need it so delete the highlighted section below. Looking at the other module note that it points to the image in your container registry. As we add more modules to the solution they will show up in this section.

```
"modules": {
  "tempSensor": {
    "version": "1.0",
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
      "image": "mcr.microsoft.com/azureiotedge-simulated-
temperature-sensor:1.0",
      "createOptions": {}
    }
  },
  "turbofanRulClassifier": {
    "version": "1.0",
    "type": "docker",
    "status": "running",
    "restartPolicy": "always",
    "settings": {
      "image": "<your registry>.azurecr.io/edgemlsample:1",
      "createOptions": {}
    }
  }
}
```

3. **Routes:** we will be working with routes quite a bit in the remainder of this portion of the walk-through. Routes define how loosely coupled modules communicate with each other. The two routes defined by the template do not match with the routing we need. The first route sends all the data from any output of the classifier to the IoT Hub (\$upstream). The other route is for tempSensor, which we just deleted. Delete the routes by deleting the highlighted area.

```
"$edgeHub": {
  "properties.desired": {
    "schemaVersion": "1.0",
```

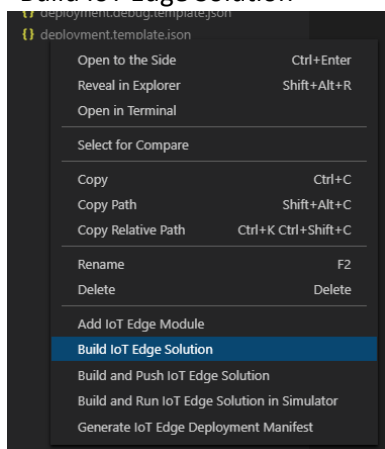


```

    "routes": {
      "turbofanRulClassifierToIoTHub": "FROM
/messages/modules/turbofanRulClassifier/outputs/* INTO
$upstream",
      "sensorToturbofanRulClassifier": "FROM
/messages/modules/tempSensor/outputs/temperatureOutput INTO
BrokeredEndpoint(\"/modules/turbofanRulClassifier/inputs/input1\"
)"
    },
    "storeAndForwardConfiguration": {
      "timeToLiveSecs": 7200
    }
  }
}

```

- **deployment.debug.template.json**: this is the debug version of deployment.template.json. We should mirror all of the changes from the deployment.template.json into this file.
 - **.env**: this file contains the username and password for accessing your registry
10. Right click on the deployment.template.json file in Visual Studio Code explorer and choose “Build IoT Edge Solution”



11. Notice that this creates a config folder with a deployment.amd64.json file. This file is the concrete deployment template for the solution.

8.3 Add Router module

Next, we add the Router module to our edge solution. The Router module handles several responsibilities for our solution:

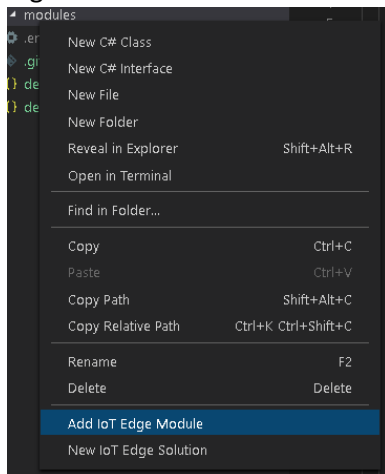
- **Receive messages from leaf devices**: as messages arrive to the edge device from downstream devices, the Router module receives the message and begins orchestrating the routing of the message.
- **Send messages to the RUL Classifier module**: when a new message is received from a downstream device, the Router module transforms the message to the format that the RUL Classifier expects. The Router sends the message to the RUL Classifier for a RUL prediction. Once the classifier has made a prediction, it sends the message back to the Router module.

- **Send RUL messages to IoT Hub:** when the Router receives messages from the classifier, it transforms the message to contain only the essential information, device ID and RUL, and sends the abbreviated message to the IoT Hub reducing the size of the messages flowing to the hub. A further refinement, which we have not done here, would send messages to the IoT Hub only when the RUL prediction falls below a threshold (e.g. $RUL < 100$ cycles). Filtering in this way, would reduce volume of messages and reduce cost of the IoT Hub.
- **Send message to the Avro Writer module:** to preserve all the data send by the downstream device, the Router module sends the entire message received from the classifier to the Avro Writer module, which will persist and upload the data using IoT Hub file upload.

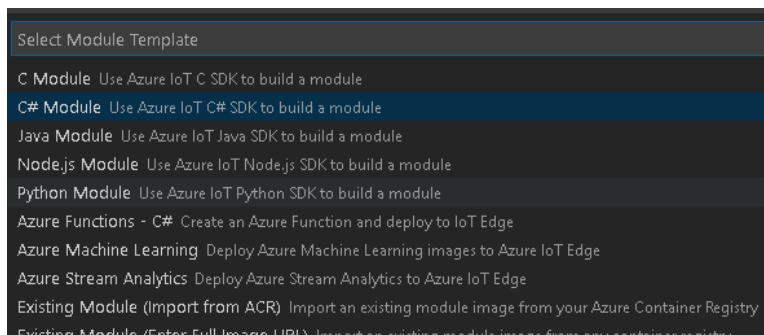
Note: the description of the module responsibilities may make the processing seem sequential, but the flow is message/event based. This is why we need an orchestration module like our Router module.

8.3.1 Create module and copy files

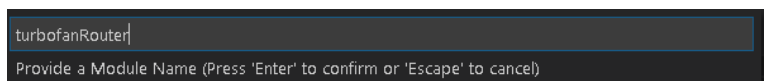
1. Right click on the modules folder in Visual Studio Code and choose 'Add IoT Edge Module'



2. Choose C# module:



3. Name the module "turbofanRouter"



- When prompted for your Docker Image Repository use the registry from the ML workspace (you can find the registry in the registryCredentials node of your *deployment.template.json* file). Note that this will be the fully qualified address to the registry like :

```
<your registry>.azurecr.io/turbofanrouter
```

```
<your registry>.azurecr.io/turbofanrouter
```

```
Provide Docker Image Repository for the Module (Press 'Enter' to confirm or 'Escape' to cancel)
```

Note: In this walk-through, we use the Azure Container Registry created by the Machine Learning service workspace, which we used to train and deploy our classifier. This is purely for convenience, we could have just as easily created a new container registry and published our modules there.

- Open a new terminal window in Visual Studio Code (Ctrl+Shift+') and copy files from the modules directory by typing the command

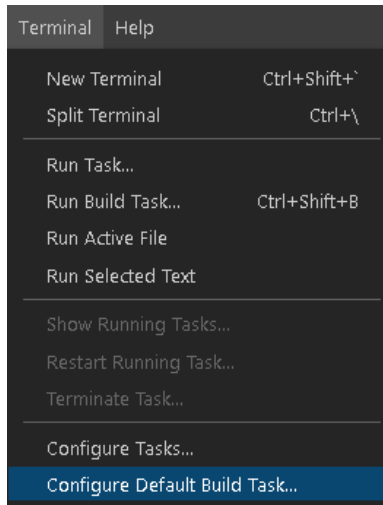
```
copy
c:\source\IoTEdgeAndMlSample\EdgeModules\modules\turbofanRouter\*.cs
c:\source\IoTEdgeAndMlSample\EdgeSolution\modules\turbofanRouter\
```

- When prompted to overwrite program.cs press 'y' and then hit Enter

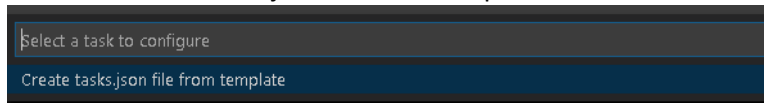
```
C:\source\IoTEdgeAndMlSample\EdgeSolution>copy c:\source\IoTEdgeAndMlSample\EdgeModules\modules\turbofanRouter\*.cs .\modules\turbofanRouter\
c:\source\IoTEdgeAndMlSample\EdgeModules\modules\turbofanRouter\Logger.cs
c:\source\IoTEdgeAndMlSample\EdgeModules\modules\turbofanRouter\Program.cs
Overwrite .\modules\turbofanRouter\Program.cs? (Yes/No/All): y
c:\source\IoTEdgeAndMlSample\EdgeModules\modules\turbofanRouter\turbofanMessage.cs
3 file(s) copied.
```

8.3.2 Build router module

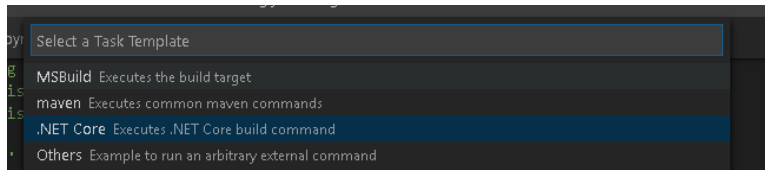
- In Visual Studio Code choose **Terminal->Configure Default Build Task**



- Click on "Create tasks.json file from template"



3. Click on “.NET Core”



4. When tasks.json opens replace the contents with:

```
{
  // See https://go.microsoft.com/fwlink/?LinkId=733558
  // for the documentation about the tasks.json format
  "version": "2.0.0",
  "tasks": [
    {
      "label": "build",
      "command": "dotnet",
      "type": "shell",
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "args": [
        "build",
        "${workspaceFolder}/modules/turbofanRouter"
      ],
      "presentation": {
        "reveal": "always"
      },
      "problemMatcher": "$msCompile"
    }
  ]
}
```

5. Save and close tasks.json
6. Run build with Ctrl+Shift+B or **Terminal->Run Build Task...**

```
> Executing task: "dotnet build" C:\source\IoTEdgeAndMISample\EdgeSolution\modules\turbofanRouter <

Microsoft (R) Build Engine version 15.9.20+g88f5fadfbc for .NET Core
Copyright (C) Microsoft Corporation. All rights reserved.

    Ctrl + click to follow link
Restore completed in 68 ms for C:\source\IoTEdgeAndMISample\EdgeSolution\modules\turbofanRouter\turbofanRouter.csproj.
turbofanRouter -> C:\source\IoTEdgeAndMISample\EdgeSolution\modules\turbofanRouter\bin\Debug\netcoreapp2.1\turbofanRouter.dll

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:01.11

Terminal will be reused by tasks, press any key to close it.
```

8.3.3 Setup module routes

As mentioned above, the Edge runtime uses routes configured in the *deployment.template.json* to manage communication between loosely coupled modules. In this section, we drill into how to set up the routes for the turbofanRouter module. We will cover the input routes first and then move on the outputs.

8.3.3.1 Inputs

1. In the Init() method of Program.cs we register 2 callbacks for the module:

```
await ioHubModuleClient.SetInputMessageHandlerAsync(  
    EndpointNames.FromLeafDevice,  
    LeafDeviceInputMessageHandler,  
    ioHubModuleClient);  
  
await ioHubModuleClient.SetInputMessageHandlerAsync(  
    EndpointNames.FromClassifier,  
    ClassifierCallbackMessageHandler,  
    ioHubModuleClient);
```

2. The first callback listens for messages sent to the "deviceInput" sink. From the diagram above, we see that we want to route messages from any leaf device to this input. In the *deployment.template.json* file we add the following route:

```
"leafMessagesToRouter": "FROM /messages/* WHERE NOT  
IS_DEFINED($connectionModuleId) INTO  
BrokeredEndpoint(\"/modules/turbofanRouter/inputs/deviceInput\")"
```

3. This statement tells the edge hub to route any message received by the edge device that was not sent by an edge module into the input called "deviceInput" on the turbofanRouter module.
4. Next add a route for messages from the rulClassifier module into the turbofanRouter module:

```
"classifierToRouter": "FROM  
/messages/modules/classifier/outputs/amloutput INTO  
BrokeredEndpoint(\"/modules/turbofanRouter/inputs/rulInput\")"
```

8.3.3.2 Outputs

1. Program.cs defines the method SendMessageToClassifier() uses the module client to send a message to the RUL classifier using the route:

```
"routerToClassifier": "FROM  
/messages/modules/turbofanRouter/outputs/classOutput INTO  
BrokeredEndpoint(\"/modules/classifier/inputs/amlInput\")"
```

2. SendRulMessageToIoTHub() uses the module client to send just the RUL data for the device to the IoT Hub via the route:

```
"routerToIoTHub": "FROM  
/messages/modules/turboFanRouter/outputs/hubOutput INTO $upstream"
```

3. SendMessageToAvroWriter() uses the module client to send the message with the RUL data added to the avroFileWriter module.

```
"routerToAvro": "FROM  
/messages/modules/turbofanRouter/outputs/avroOutput INTO  
BrokeredEndpoint(\"/modules/avroFileWriter/inputs/avroModuleInput\")"
```

4. HandleBadMessage() sends failed messages upstream the IoT Hub where they can be routed for later

```
"deadLetter": "FROM  
/messages/modules/turboFanRouter/outputs/deadMessages INTO $upstream"
```

8.3.3.3 Summary

With all the routes taken together your “\$edgeHub” node should look like the following (changes highlighted)

```
"$edgeHub": {
  "properties.desired": {
    "schemaVersion": "1.0",
    "routes": {
      "leafMessagesToRouter": "FROM /messages/* WHERE NOT
IS_DEFINED($connectionModuleId) INTO
BrokeredEndpoint(\"/modules/turbofanRouter/inputs/deviceInput\")",
      "classifierToRouter": "FROM
/messages/modules/turbofanRulClassifier/outputs/amlOutput INTO
BrokeredEndpoint(\"/modules/turbofanRouter/inputs/rulInput\")",
      "routerToClassifier": "FROM
/messages/modules/turbofanRouter/outputs/classOutput INTO
BrokeredEndpoint(\"/modules/turbofanRulClassifier/inputs/amlInput\")",
      "routerToIoTHub": "FROM
/messages/modules/turboFanRouter/outputs/hubOutput INTO $upstream",
      "routerToAvro": "FROM
/messages/modules/turbofanRouter/outputs/avroOutput INTO
BrokeredEndpoint(\"/modules/avroFileWriter/inputs/avroModuleInput\")",
      "deadLetter": "FROM
/messages/modules/turboFanRouter/outputs/deadMessages INTO $upstream"
    },
    "storeAndForwardConfiguration": {
      "timeToLiveSecs": 7200
    }
  }
}
```

Note: Adding the turbofanRouter module created a route like:

```
"turbofanRouterToIoTHub": "FROM
/messages/modules/turbofanRouter/outputs/* INTO $upstream".
```

Remove this route, leaving only the routes listed above in your deployment.template.json file.

8.3.3.4 Copy routes to deployment.debug.template.json

As a final step, to keep our files in sync mirror the changes you made to deployment.template.json in deployment.debug.template.json

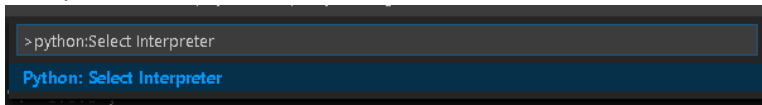
8.4 Add Avro Writer module

The Avro Writer module has two responsibilities in our solution, store messages and upload files.

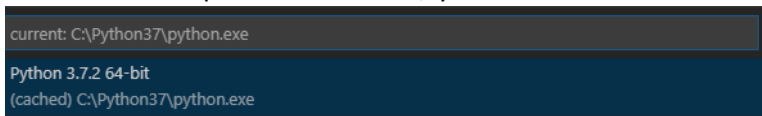
- **Store messages:** when the Avro Writer module receives a message, it writes the message to the local file system in Avro format. We use a bind mount, which mounts a directory (in this case /data/avrofiles) into a path in the module's container. This allows the module to write to a local path (/avrofiles) and have those files accessible directly from the edge device.
- **Upload files:** the Avro Writer module uses the Azure IoT Hub file upload feature to upload files to an Azure storage account. Once a file is successfully uploaded the module deletes the file from disk

8.4.1 Create module and copy files

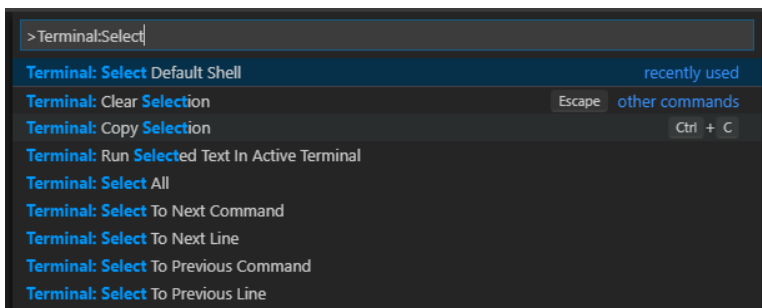
1. Open command palette and type "Python: Select Interpreter" then click on 'Python: Select Interpreter'



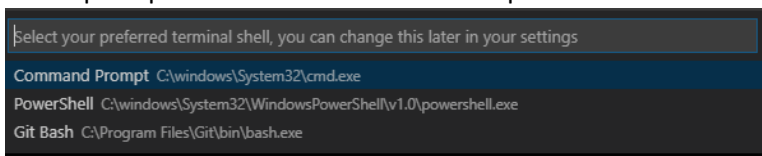
2. Choose the interpreter found in c:\Python37



3. Open the command palette again and type: "Terminal: Select" and choose "Terminal: Select Default Shell"



4. When prompted choose "Command Prompt"

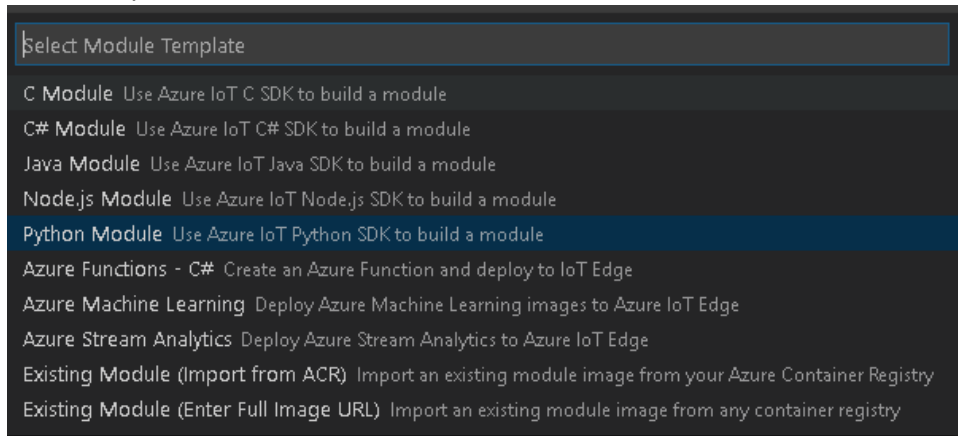


5. Open a new terminal shell (Ctrl+Shift+`) or **Terminal->New Terminal**
6. Install the Python module template:

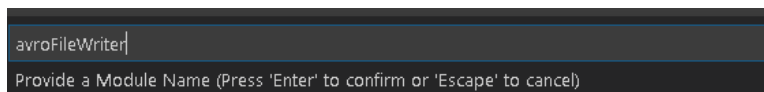
```
pip install --upgrade --user cookiecutter
```

7. Right click on the modules folder in Visual Studio Code and choose "Add IoT Edge Module"

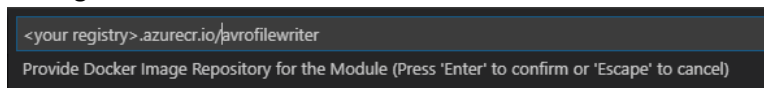
8. Choose "Python Module"



9. Name the module "avroFileWriter"



10. When prompted for your Docker Image Repository use the same registry as you used when adding the Router module



11. Copy files from the sample module into the solution. In the Visual Studio Code terminal window run

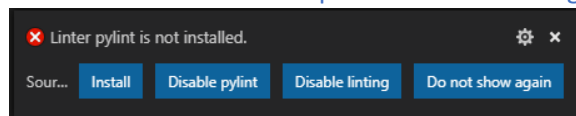
```
copy
C:\source\IoTEdgeAndMlSample\EdgeModules\modules\avroFileWriter\*.py
C:\source\IoTEdgeAndMlSample\EdgeSolution\modules\avroFileWriter\
```

12. If prompted to overwrite main.py type 'y' and then hit Enter

```
C:\source\IoTEdgeAndMlSample\EdgeSolution>copy C:\source\IoTEdgeAndMlSample\EdgeModules\modules\avroFile
es\avroFileWriter\
C:\source\IoTEdgeAndMlSample\EdgeModules\modules\avroFileWriter\configreader.py
C:\source\IoTEdgeAndMlSample\EdgeModules\modules\avroFileWriter\filemanager.py
C:\source\IoTEdgeAndMlSample\EdgeModules\modules\avroFileWriter\main.py
Overwrite C:\source\IoTEdgeAndMlSample\EdgeSolution\modules\avroFileWriter\main.py? (Yes/No/All): y
C:\source\IoTEdgeAndMlSample\EdgeModules\modules\avroFileWriter\schema.py
C:\source\IoTEdgeAndMlSample\EdgeModules\modules\avroFileWriter\uploader.py
5 file(s) copied.
```

13. Notice that filemanager.py and schema.py have been added to the solution and main.py has been updated

Note: when you open a Python file you may be prompted to install pylint. You do not need to install the linter to complete this walk-through, but you may if you choose



8.4.2 Bind mount for data files

As mentioned in the intro, the writer module relies on the presence of bind mount to write Avro files to the edge device's file system.

8.4.2.1 Add directory to edge device

1. Connect to your edge device VM using ssh

```
ssh -l <user>@IoTEdge-<extension>.<region>.cloudapp.azure.com
```

2. Create the directory

```
sudo mkdir -p /data/avrofiles
```

3. Update directory permissions to make it writeable by the container

```
sudo chmod ugo+rw /data/avrofiles
```

4. Validate the directory now has write (w) permission for user, group and owner.

```
ls -la /data
```

```
root@IoTEdge-1:~# ls -la /data
total 12
drwxr-xr-x  3 root root 4096 Jan 19 23:09 .
drwxr-xr-x 25 root root 4096 Jan 20 19:00 ..
drwxrwxrwx  2 root root 4096 Jan 19 23:09 avrofiles
root@IoTEdge-1:~#
```

8.4.2.2 Add directory to the module

To add the directory to the module's container we will modify the Dockerfiles associated with the avroFileWriter module. You will note, that there are 3 Dockerfiles associated with the module, Dockerfile.amd64, Dockerfile.amd64.debug, and Dockerfile.arm32v7. These files should be kept in sync in case we wish to debug or deploy to an arm32 device. For the walk-through we'll focus only on Dockerfile.amd64.

1. On your development machine open the files Dockerfile.amd64
2. Modify the file so that it looks like:

```
FROM ubuntu:xenial

WORKDIR /app

RUN apt-get update && \
    apt-get install -y --no-install-recommends libcurl4-openssl-dev \
    python3-pip libboost-python1.58-dev libpython3-dev && \
    rm -rf /var/lib/apt/lists/*

RUN pip3 install --upgrade pip
COPY requirements.txt ./
RUN pip install -r requirements.txt

COPY . .

RUN useradd -ms /bin/bash moduleuser
RUN mkdir /avrofiles \
    && chown moduleuser /avrofiles
```

```
USER moduleuser
```

```
CMD "python3", "-u", "./main.py"
```

3. These 2 lines instruct the Docker build process to create a top-level directory called /avrofiles in the image and then to make the moduleuser the owner of that directory. It is important that these commands are inserted after the module user is added to the image with the useradd command and before the context switches to the moduleuser (USER moduleuser)
4. Make the corresponding changes to Dockerfile.amd64.debug and Dockerfile.arm32v7

8.4.2.3 Update the module configuration

The final step of creating the bind is to update the deployment.template.json (and deployment.debug.template.json) files with the bind information.

1. Open deployment.template.json
2. Modify the module definition for avroFileWriter by adding the lines below, which points the container directory /avrofiles to the local directory on the edge device.

```
"avroFileWriter": {  
  "version": "1.0",  
  "type": "docker",  
  "status": "running",  
  "restartPolicy": "always",  
  "settings": {  
    "image": "${MODULES.avroFileWriter}",  
    "createOptions": {  
      "HostConfig": {  
        "Binds": [  
          "/data/avrofiles:/avrofiles"  
        ]  
      }  
    }  
  }  
}
```

3. Make the corresponding changes to deployment.debug.template.json

8.4.3 Bind mount for access to config.yaml

We need to add one more bind for the writer module. This bind gives the module access to read the connection string from the /etc/iotedge/config.yaml file on the edge device. We need the connection string to create an IoT HubClient so that we can call the upload_blob_async method to upload files to the IoT Hub. The steps for adding this bind are similar to the ones above

8.4.3.1 Update directory permission on edge device

1. Open edge device using ssh

```
ssh -l <user>@IoTEdge-<extension>.<region>.cloudapp.azure.com
```

2. Add read permission to the config.yaml file

```
sudo chmod +r /etc/iotedge/config.yaml
```

3. Ensure permission are correct

```
ls -la /etc/iotedge/
```

```
@IoTEdge-~$ ls -la /etc/iotedge/
total 24
drwxr-xr-x  2 root    root    4096 Jan 19 00:07 .
drwxr-xr-x 99 root    root    4096 Jan 19 23:10 ..
-rwxr-xr-x  1 root    root     805 Nov 14 00:33 configedge.sh
-r--r--r--  1 iotedge iotedge 6424 Jan 19 00:07 config.yaml
-rwxr-xr-x  1 root    root    1588 Nov 14 00:32 installedge.sh
@IoTEdge-~$
```

8.4.3.2 Add directory to module

1. On your development machine, open the files Dockerfile.amd64
2. Modify the file so that it looks like:

```
FROM ubuntu:xenial

WORKDIR /app

RUN apt-get update && \
    apt-get install -y --no-install-recommends libcurl4-openssl-dev \
    python3-pip libboost-python1.58-dev libpython3-dev && \
    rm -rf /var/lib/apt/lists/*

RUN pip3 install --upgrade pip
COPY requirements.txt ./
RUN pip install -r requirements.txt

COPY . .

RUN useradd -ms /bin/bash moduleuser
RUN mkdir /avrofiles \
    && chown moduleuser /avrofiles
RUN mkdir -p /app/iotconfig \
    && chown moduleuser /app/iotconfig

USER moduleuser

CMD "python3", "-u", "./main.py"
```

3. Make the corresponding changes to Dockerfile.amd64.debug and Dockerfile.arm32v7

8.4.3.3 Update the module configuration

1. Open deployment.template.json
2. Modify the module definition for avroFileWriter by adding the line below, which points the container directory /app/iotconfig to the local directory /etc/iotedge on the edge device

```
"avroFileWriter": {
  "version": "1.0",
  "type": "docker",
  "status": "running",
  "restartPolicy": "always",
  "settings": {
    "image": "${MODULES.avroFileWriter}",
    "createOptions": {
```

```

        "HostConfig": {
            "Binds": [
                "/data/avrofiles:/avrofiles",
                "/etc/iotedgedge:/app/iotconfig"
            ]
        }
    }
}

```

3. Make the corresponding changes to deployment.debug.template.json

8.5 Install dependencies

The writer module takes a dependency on two Python libraries, fastavro and PyYAML. We need to install the dependencies on our development machine and instruct the Docker build process to install them in our module's image.

8.5.1 PyYAML

1. On your development machine open requirements.txt and add pyyaml making the file

```

azure-iot-hub-device-client~=1.4.3
pyyaml

```

2. Open Dockerfile.amd64 and add a line to upgrade setuptools

```

FROM ubuntu:xenial

WORKDIR /app

RUN apt-get update && \
    apt-get install -y --no-install-recommends libcurl4-openssl-dev \
    python3-pip libboost-python1.58-dev libpython3-dev && \
    rm -rf /var/lib/apt/lists/*

RUN pip3 install --upgrade pip
RUN pip install -U pip setuptools
COPY requirements.txt ./
RUN pip install -r requirements.txt

COPY . .

RUN useradd -ms /bin/bash moduleuser
RUN mkdir /avrofiles \
    && chown moduleuser /avrofiles
RUN mkdir -p /app/iotconfig \
    && chown moduleuser /app/iotconfig
USER moduleuser

CMD [ "python3", "-u", "./main.py" ]

```

3. Make the corresponding changes to Dockerfile.amd64.debug and Dockerfile.arm32v7
4. Install pyyaml locally by opening a terminal in Visual Studio Code and typing

```
pip install pyyaml
```

```
C:\source\IoTEdgeAndMISample\DevVM>pip install pyyaml
Collecting pyyaml
  Downloading https://files.pythonhosted.org/packages/bf/96/d02ef8e1f3073e07ffdc24
0444e5041f403f29c0775f9f1653f18221082f/PyYAML-3.13-cp37m-win_amd64.whl (206kB
)
  100% |#####| 215kB 4.7MB/s
Installing collected packages: pyyaml
Successfully installed pyyaml-3.13
```

8.5.2 Fastavro

1. Open requirements.txt and add fastavro making the file contents:

```
azure-iot-hub-device-client~=1.4.3
pyyaml
fastavro
```

2. Install fastavro to your development machine. Using Visual Studio Code terminal

```
pip install fastavro
```

```
C:\source\IoTEdgeAndMISample\EdgeSolution>pip install fastavro
Collecting fastavro
  Downloading https://files.pythonhosted.org/packages/ef/6c/c229bc49369911f517ad0161acf0a5bc818
  Downloading https://files.pythonhosted.org/packages/ef/6c/c229bc49369911f517ad0161acf0a5bc818
  Downloading https://files.pythonhosted.org/packages/ef/6c/c229bc49369911f517ad0161acf0a5bc818
  Downloading https://files.pythonhosted.org/packages/ef/6c/c229bc49369911f517ad0161acf0a5bc818
  Downloading https://files.pythonhosted.org/packages/ef/6c/c229bc49369911f517ad0161acf0a5bc818
1 (296kB)
  100% |#####| 296kB 4.5MB/s
Installing collected packages: fastavro
Successfully installed fastavro-0.21.16
```

8.6 Reconfigure IoT Hub

By introducing the edge device and modules to the system, we have changed our expectations about what data will be sent to the hub and for what purpose. We need to reconfigure the routing in the hub to deal with our new reality.

Note: we reconfigure the hub before deploying modules because some of the hub settings, specifically file upload, needs to be correctly set up for the avroFileWriter module to run correctly

8.6.1 Set up route for RUL messages in IoT Hub

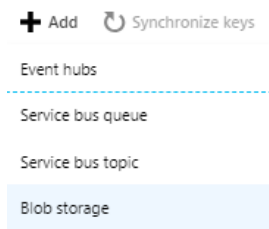
With the router and classifier in place, we expect to receive regular messages containing only the device id and the RUL prediction for the device. We want to route the RUL data to its own storage location where we can monitor the status of the devices, build reports and fire alerts as needed. At the same time, we want any device data that is still being sent directly by a leaf device that has not yet been attached to our edge device to continue to route to the current storage location.

8.6.1.1 Create a RUL storage endpoint

First, we create an endpoint where we will send the RUL data as it is received

1. Log into <http://portal.azure.com> and navigate to your IoT Hub
2. From the left navigation choose “Message routing”

3. Click on "Custom Endpoints"
4. Click "+ Add" and choose "BLOB Storage"



5. Name the endpoint "ruldata"
6. Click on "Pick a container"
7. Choose the storage account you have used throughout this walk-through
8. Click "+ Container", name the container "ruldata" then click "OK"

9. Select the "ruldata" container you just created and hit "Select"
10. Back on the "Add a storage endpoint page" click "Create"

8.6.1.2 Create RUL message route

Now we route messages with RUL data to our new endpoint

1. In the IoT Hub "Message routing" page click on "Routes"
2. Click on "+Add"
3. Name the route "RulMessagesRoute" and select the "ruldata" Endpoint

4. For the 'Routing query' enter:

```
IS_DEFINED($body.PredictedRul) AND
NOT IS_DEFINED($body.OperationalSetting1)
```

Routing query

```
1 IS_DEFINED($body.PredictedRul) AND
2 NOT IS_DEFINED($body.OperationalSetting1)
```

- Expand 'Test' and the 'Message body' and replace the message with (an example of our expected messages)

```
{
  "ConnectionDeviceId": "aaLeafDevice_1",
  "CorrelationId": "b27e97bb-06c5-4553-a064-e9ad59c0fdd3",
  "PredictedRul": 132.62721409309165,
  "CycleTime": 64.0
}
```

^ Test

A sample message tests your route query. Results will show whether the sample matched the query or not, and v

✓ System properties

✓ Application properties

^ Message body

Message body queries require JSON. Modify this sample to simulate messages from your devices.

```
1 {
2   "ConnectionDeviceId": "aaLeafDevice_1",
3   "CorrelationId": "b27e97bb-06c5-4553-a064-e9ad59c0fdd3",
4   "PredictedRul": 132.62721409309165,
5   "CycleTime": 64.0
6 }
```

- Scroll down and click on “Test Route” to validate that the query will route messages that have "PredictedRul" in the message body.

[Test route](#) The message matched the query.

- Click “Save”

8.6.1.3 Update turbofanDeviceToStorage route

We don't want to route the new prediction data to our old storage location so update the route to prevent it.

- In the IoT Hub “Message routing” page click on “Routes”
- Click "turbofanDeviceDataToStorage" (or whatever name you gave to your initial device data route”

- Update the routing query to

```
IS_DEFINED($body.OperationalSetting1)
```

- Open “Test” and “Message body” and add the message

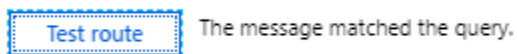
```
{
  "Sensor13": 2387.96,
  "OperationalSetting1": -0.0008,
  "Sensor6": 21.61,
  "Sensor11": 47.2,
  "Sensor9": 9061.45,
  "Sensor4": 1397.86,
  "Sensor14": 8140.39,
  "Sensor18": 2388.0,
  "Sensor12": 522.87,
}
```

```

"Sensor2": 642.42,
"Sensor17": 391.0,
"OperationalSetting3": 100.0,
"Sensor1": 518.67,
"OperationalSetting2": 0.0002,
"Sensor20": 39.03,
"DeviceId": 19.0,
"Sensor5": 14.62,
"PredictedRul": 212.00132402791962,
"Sensor8": 2388.01,
"Sensor16": 0.03,
"CycleTime": 42.0,
"Sensor21": 23.3188,
"Sensor15": 8.3773,
"Sensor3": 1580.09,
"Sensor10": 1.3,
"Sensor7": 554.57,
"Sensor19": 100.0
}

```

5. Click on "Test route"

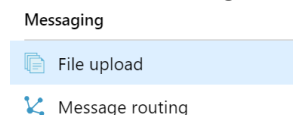


6. Click "Save"

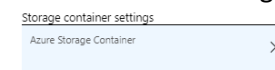
8.6.2 Configure file upload

The IoT Hub file upload feature needs to be configured to enable the file writer module to upload files to storage.

1. Log into <http://portal.azure.com> and navigate to your IoT Hub
2. From the left navigator choose "File upload"



3. Click on "Azure Storage Container"



4. Select your storage account from the list
5. In "Containers" click on "+ Container"

6. Name the container “uploadturbobanfiles” and click “OK”

Containers
turbodevicedata

+ Container Refresh

New container

* Name

uploadturbobanfiles ✓

Public access level ⓘ

Private (no anonymous access) ▼

OK

Cancel

7. Click on “uploadturbobanfiles” from the list of containers and click “Select”

+ Container Refresh

NAME

devicedata

ruldata

uploadturbobanfiles

Select

8. Click on 'Save'; the portal will notify you when the save is complete

Note: We aren't turning on upload notification for this walk-through, but see <https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-java-java-file-upload#receive-a-file-upload-notification> for details on how to handle file upload notification.

8.7 Build, publish and deploy modules

Now that we have made the configuration changes, we are ready to build the images and publish them to our Azure Container Registry. The build process uses the deployment.template.json file to determine which modules need to be built. The settings for each module, including version, is found in the module.json file in the module folder. The build will first run a Docker build on the Dockerfiles matching the current configuration found in the module.json file to create an image. Then the build will publish the image to the registry from the module.json file with a version tag matching the one in the module.json file. Finally, the build will produce a configuration specific deployment template (e.g. deployment.amd64.json), which we will deploy to the IoT Hub edge device. The edge device will read the information from the IoT Hub, and based on the instructions will download the modules, configure the routes, set any desired properties. This deployment method has 2 side effects that you should be aware of

1. **Deployment lag:** since the Edge runtime has to recognize the change to the hub before it starts to reconfigure, it can take some amount of time after you deploy your modules until the runtime picks them up and starts to update the edge device
2. **Module versions matter:** if you publish a new version of a module's container to your container registry using the same version tags as the previous module the runtime will not download the new version of the module. It does a comparison of the version tag of the local image and the desired image from the deployment manifest, if those versions match the runtime takes no further action. It is, therefore, important to bump the version of your module each time you wish to deploy new changes. Do this by changing the "version" property under the "tag" property in the module.json file for the module you are changing. Then build and publish the module.

```
{
  "$schema-version": "0.0.1",
  "description": "",
  "image": {
    "repository": "turbofanacrptzrofnh.azurecr.io/avrofilewriter",
    "tag": {
      "version": "0.0.1",
      "platforms": {
        "amd64": "./Dockerfile.amd64",
        "amd64.debug": "./Dockerfile.amd64.debug",
        "arm32v7": "./Dockerfile.arm32v7"
      }
    },
    "buildOptions": []
  },
  "language": "python"
}
```

8.7.1 Build and publish

1. In Visual Studio Code on your development VM
2. Open the .env file in your solution, it will look like:

```
CONTAINER_REGISTRY_USERNAME_<your registry name>=<ACR username>
CONTAINER_REGISTRY_PASSWORD_<your registry name>=<ACR password>
```

3. Open the deployment.template.json file and find the "registryCredentials" property

```
"registryCredentials": {
  "turbofanacrptzrofnh": {
    "username": "$CONTAINER_REGISTRY_USERNAME_<your registry
name>",
    "password": "$CONTAINER_REGISTRY_PASSWORD_<your registry
name>",
    "address": "<ACR login server>"
  }
}
```

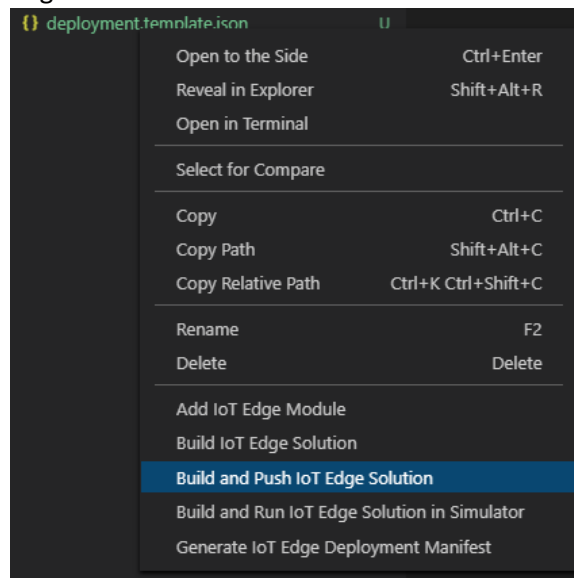
4. Open a Visual Studio Code terminal window and login to your container registry with the command

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

```
C:\source\IoTEdgeAndMISample\EdgeSolution>docker login -u turbofanacrgptcrnfeib -p 86K21agX7gkx5t8uM9b-FH20hp73Uwq turbofanacrgptcrnfeib.azurecr.io
WARNING! Using --password via the CLI is insecure. Use --password-stdin.
Login Succeeded

C:\source\IoTEdgeAndMISample\EdgeSolution>
```

5. In Visual Studio Code right-click on deployment.template.json and choose “Build and Push IoT Edge Solution”



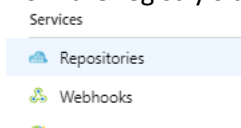
8.7.2 View modules in the registry

Once the build successfully completes, we will be able to use the Azure Portal to review our published modules

1. Open <http://portal.azure.com>
2. Navigate to your Machine Learning service workspace and click the hyperlink for “Registry”

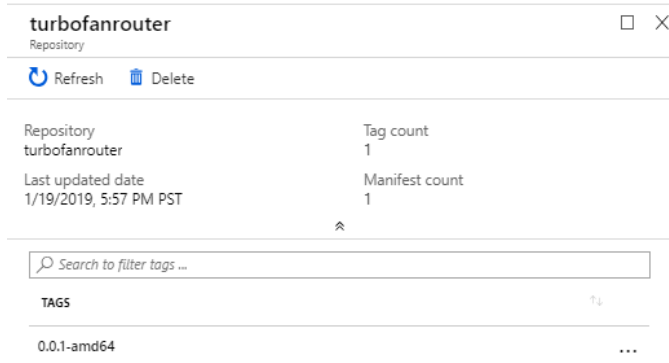
Resource group :	turbofanDemo	Storage :	turbofanstoragebwkafovr
Location :	West US 2	Registry :	turbofanacrguqalfee
Subscription :	407388334637	Key Vault :	turbofankeyvaultyhunuhtt
Subscription ID :	407388334637-407388334637-407388334637	Application Insights :	turbofaninsightssfsoksxb

3. From the registry side navigator click “Repositories”



4. Note that “avrofilewriter” and “turbofanrouter” appear as repositories

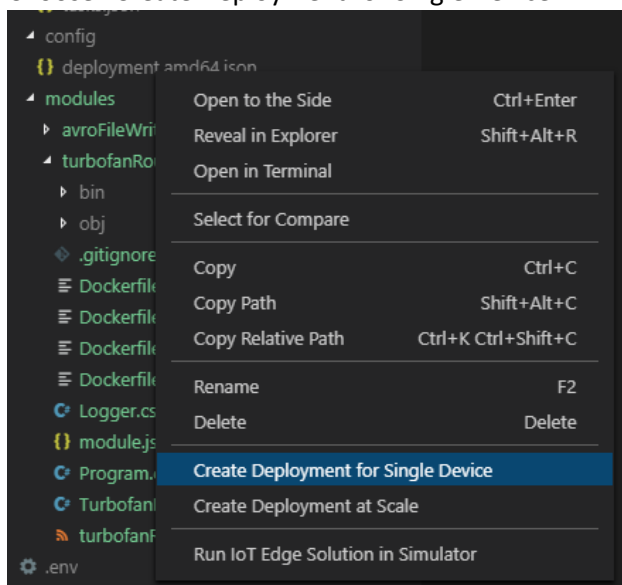
- Click on “turbofanrouter” and note that you have published one image tagged as 0.0.1-amd64



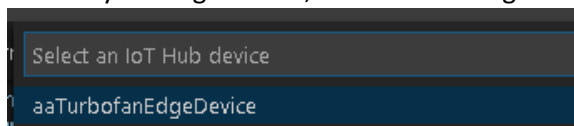
8.7.3 Deploy modules to IoT Edge device

We have built and configured the modules in our solution, now we will deploy the modules to edge device.

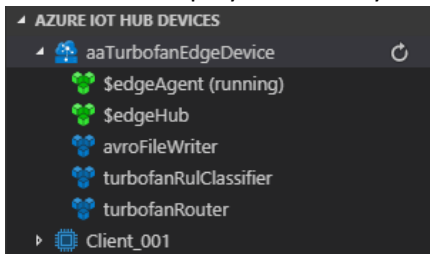
- In Visual Studio Code right click on the file deployment.amd64.json in the config folder
- Choose “Create Deployment for Single Device”



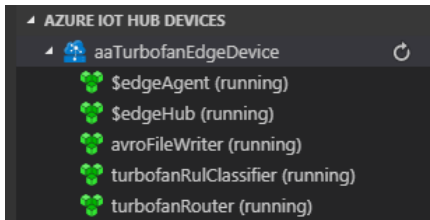
- Choose your edge device, "aaTurboFanEdgeDevice"



4. Refresh the 'AZURE IOT HUB DEVICES' panel in Visual Studio Code explorer, you will see the 3 modules are deployed but not yet running.



5. Refresh again after several minutes and you will see the modules running.



It can take several minutes for the modules to get installed and running on the edge device and settle into a steady running state. During that time you may see modules start and stop as they try to establish a connection with the hub module.

8.8 Diagnosing failures

In this section we share a few techniques for understanding what has gone wrong with a module or modules. Often a failure can first be spotted from the status in the Visual Studio Code.

8.8.1 Identify failed modules

- **Visual Studio Code:** Look at the “AZURE IOT HUB DEVICES” panel if most modules are in a running state, but one is stopped you need to investigate that module further. If all modules are in a stopped state for a long period of time this may indicate failure as well.
- **Azure Portal:** By navigating to the IoT Hub in the portal and then finding the device details page (under IoT Edge, drill into your device) you may find that a module has reported an error or has never reported anything to the IoT Hub.

8.8.2 Diagnosing from the edge device

By logging into the edge device, you can gain access to a good deal of information about the status of your modules. The main mechanism we use are the Docker commands that let us examine the containers and images on the device.

1. List all running containers, we expect to see a container for each module with a name that corresponds to the module. Also, this lists the exact image for the container including version so you can match with your expectation. You can also list images by substituting “image” for “container” in the command

```
sudo docker container ls
```

```
markmyd@IoTEdge-nhxxhmg:~$ sudo docker container ls
CONTAINER ID        IMAGE                                     COMMAND                  CREATED             STATUS              NAMES
a869a5391842       mcr.microsoft.com/azureiotedge-hub:1.0  "/bin/sh -c 'echo \"$..." 39 minutes ago      Up 39 minutes      edgeHub
57d19b1a6627       turbofanacrtzrofnh.azurecr.io/turbofanrouter:0.0.1-amd64  "dotnet turbofanRout..." 39 minutes ago      Up 39 minutes      turbofanRouter
2229c923cdd2       turbofanacrtzrofnh.azurecr.io/edgемisample:1              "runsvdir /var/runit"    39 minutes ago      Up 39 minutes      turbofanRulClassifier
88fa671a5109       turbofanacrtzrofnh.azurecr.io/avrofilewriter:0.0.1-amd64  "python3 -u ./main.py"   44 minutes ago      Up 43 minutes      avroFileWriter
cf7f47da0c22       mcr.microsoft.com/azureiotedge-agent:1.0  "/bin/sh -c 'echo \"$..." 29 hours ago        Up About an hour    edgeAgent
```

2. Get the logs for a container. This outputs whatever has been written to StdErr and StdOut in the container. This command will work for containers that have started and then died for some reason. It is also useful for understanding what has been happening with the edgeAgent or edgeHub

```
sudo docker container logs <container name>
```

```
markmyd@IoTEdge-nhxxhmg:~$ sudo docker container logs edgeHub
[2019-01-20 04:33:10 : Starting Edge Hub
[01/20/2019 04:33:10.560 AM] Edge Hub Main()
[01/20/2019 04:33:11.749 AM] Found intermediate certificates: [CN=iotedged workload ca:02/15/2019 05:15:20],[CN=tu
2019 05:15:18]
2019-01-20 04:33:13.058 +00:00 [INF] - Created persistent store at /tmp/edgeHub
2019-01-20 04:33:13.109 +00:00 [INF] - Starting Edge Hub
2019-01-20 04:33:13.109 +00:00 [INF] -
AZURE
IOT EDGE
2019-01-20 04:33:13.113 +00:00 [INF] - Version - 1.0.5.19141174 (d76e0316c6f324345d77c48a83ce836d09392699)
2019-01-20 04:33:13.117 +00:00 [INF] - Loaded server certificate with expiration date of "2019-02-15T05:15:20.000
2019-01-20 04:33:13.151 +00:00 [INF] - Created new message store
2019-01-20 04:33:13.221 +00:00 [INF] - Created device scope identities cache
2019-01-20 04:33:13.222 +00:00 [INF] - Started task to cleanup processed and stale messages
2019-01-20 04:33:13.250 +00:00 [INF] - Starting refresh of device scope identities cache
2019-01-20 04:33:13.382 +00:00 [INF] - Initializing configuration
2019-01-20 04:33:13.408 +00:00 [INF] - New device connection for device aaTurbofanEdgeDevice/$edgeHub
```

3. Inspect a container. This gives a ton of information about the image. The data can be filtered depending on what you are looking for. As an example, if you want to see if the binds on the avroFileWriter are correct you can use the command

```
sudo docker container inspect -f "{{ json .Mounts }}"
avroFileWriter | python -m json.tool
```

```
markmyd@IoTEdge-nhxxhmg:~$ sudo docker container inspect -f "{{ json .Mounts }}" avroFileWriter | python -m json.tool
[
  {
    "Destination": "/avrofiles",
    "Mode": "",
    "Propagation": "rprivate",
    "RW": true,
    "Source": "/data/avrofiles",
    "Type": "bind"
  },
  {
    "Destination": "/app/iotconfig",
    "Mode": "",
    "Propagation": "rprivate",
    "RW": true,
    "Source": "/etc/iotedge",
    "Type": "bind"
  }
]
```

4. Connect to a running container. This can be helpful if you want to examine the container while it is running:

```
sudo docker exec -it avroFileWriter bash
markmyd@IoTEdge-nhxxhmg:~$ sudo docker exec -it avroFileWriter bash
moduleuser@80fa671a5109:/app$
```

8.9 Summary

In this section, we created an IoT Edge Solution in Visual Studio Code with three modules, a classifier, a router, and a file writer/uploader. We set up the routes to allow the modules to communicate with each other on the edge device, modified the configuration of the edge device, and updated the Dockerfiles to install dependencies and add bind mounts to the modules' containers. Next, we updated the configuration of the IoT Hub to route our messages based on type and to handle file uploads. With everything in place, we deployed the modules to the IoT Edge device and ensured the modules were running correctly.

More information can be found at the following pages:

1. [Deploy Azure Machine Learning as an IoT Edge module](#)
2. [Develop a C# IoT Edge module and deploy to your simulated device](#)
3. [Develop and deploy a Python IoT Edge module to your simulated device](#)
4. [Deploy your first IoT Edge module to a Linux x64 device](#)
5. [Learn how to deploy modules and establish routes in IoT Edge](#)
6. [IoT Hub message routing query syntax](#)
7. [IoT Hub message routing: now with routing on message body](#)
8. [Upload files with IoT Hub](#)
9. [Upload files from your device to the cloud with IoT Hub](#)

9 Send device data

9.1 Introduction

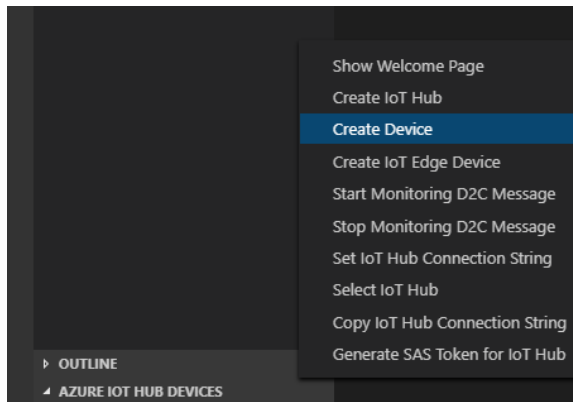
In this section, we will use our development machine as a simulated leaf device sending data to the edge device we configured above. The structure of the leaf device is like that of a single device within the device harness, which we used to generate our initial data set. Of course, the leaf device could be running on a separate device or VM instance.

We will monitor the operation of the edge device while the leaf device is sending data. Once the leaf device is finished running, we will take a look at the data in our storage account to validate everything worked as expected.

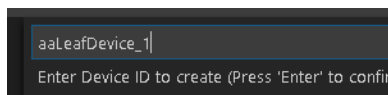
9.2 Configure leaf device

We need create the device in our Azure IoT Hub. Perform all the steps from your development machine.

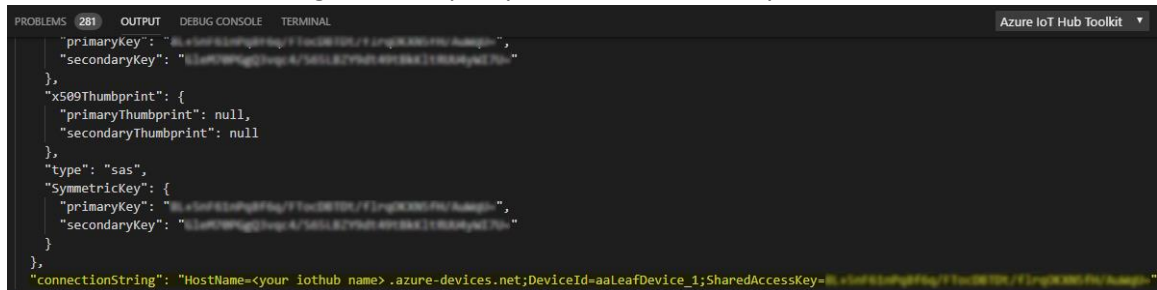
1. In Visual Studio Code click the “...” on the “AZURE IOT HUB DEVICES” pane and choose “Create Device”



2. Give the client a name, "aaLeafDevice_1". Once again, we use the "aa" prefix for sorting convenience



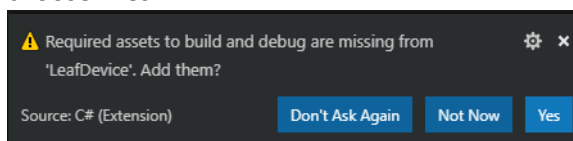
3. Note the connection string in the output, you will need it when you run the device



9.3 Build and run leaf device

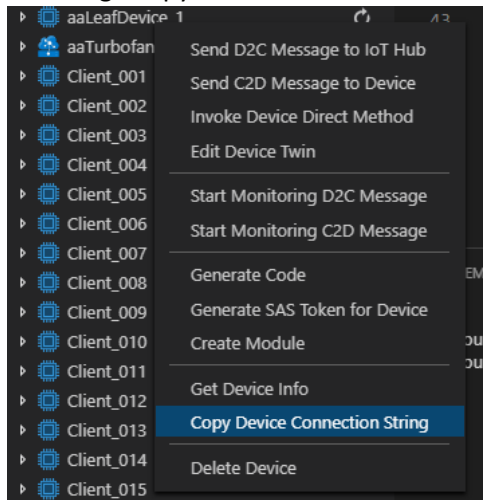
Open the LeafDevice project, build it and then run it. All steps are run on the development machine. The LeafDevice

1. In Visual Studio Code **File->Open Folder...** and open C:\source\IoTEdgeAndMISample\LeafDevice
2. Visual Studio Code will start and initialize extensions. If you are prompted to add required assets choose “Yes”



3. Build the project (Ctrl+Shift+B) or **Terminal->Run Build Task...** and select “build” from the dialog

- Find the device connection string, either captured above or by right clicking the device and selecting “Copy Device Connection String”



- Find the fully qualified domain name (FQDN) for your edge device, this will be the same as the hostname you set in the Edge device [configuration step](#). Find the FQDN by navigating to your edge device virtual machine in the portal and copying the “DNS name” from the overview.
- Run the leaf device application with the command below substituting your device connection string from step 4 and your FQDN from step 5. Also pass the location of your certificate, which the leaf device will use to validate its connection with your edge device

```
dotnet run -- -x
"<device_connection_string>;GatewayHostName=<edge_device_fqdn>" -c
C:\edgeCertificates\certs\azure-iot-test-only.root.ca.cert.pem
```

- The application will attempt to install the certificate onto your development machine, when it does you will need to accept the security warning



8. You will see device data being sent to your edge device

```
1/21/2019 3:49:09 AM > Sending message: {"DeviceId":21,"CycleTime":19,"OperationalSetting1":15,"Sensor2":604.88,"Sensor3":1498.84,"Sensor4":1304.33,"Sensor5":10.52,"Sensor6":15.49,"Sensor12":371.54,"Sensor13":2388.15,"Sensor14":8133.23,"Sensor15":8.67,"Sensor16":0.03,"Sensor4"}
1/21/2019 3:49:10 AM > Sending message: {"DeviceId":21,"CycleTime":20,"OperationalSetting1":2Sensor2":536.56,"Sensor3":1267.25,"Sensor4":1054.86,"Sensor5":7.05,"Sensor6":9.03,"Sensor7":1sor12":164.82,"Sensor13":2028.2,"Sensor14":7875.78,"Sensor15":10.9022,"Sensor16":0.02,"Sensor}
1/21/2019 3:49:11 AM > Sending message: {"DeviceId":21,"CycleTime":21,"OperationalSetting1":2Sensor2":537.07,"Sensor3":1264.96,"Sensor4":1049.84,"Sensor5":7.05,"Sensor6":9.03,"Sensor7":1sor12":164.35,"Sensor13":2028.3,"Sensor14":7883.93,"Sensor15":10.9402,"Sensor16":0.02,"Sensor}
1/21/2019 3:49:12 AM > Sending message: {"DeviceId":21,"CycleTime":22,"OperationalSetting1":15,"Sensor2":604.16,"Sensor3":1498.81,"Sensor4":1311.84,"Sensor5":10.52,"Sensor6":15.49,"Sensor12":371.31,"Sensor13":2388.13,"Sensor14":8129.32,"Sensor15":8.6632,"Sensor16":0.03,"Sensor7.1383}
```

9.4 Check output

9.4.1 Edge machine output

The output from the avroFileWriter module can be readily observed by looking at the edge device.

1. ssh into your edge machine
2. Look for files written to disk by running, you should see the file

```
ls -ltr /data/avrofiles/
root@IoTEdge-nhxxhmg:~$ ls -ltr /data/avrofiles/
/data/avrofiles/:
total 4
drwxr-xr-x 3 markayd markayd 4096 Jan 21 04:02 2019

/data/avrofiles/2019:
total 4
drwxr-xr-x 3 markayd markayd 4096 Jan 21 04:02 1

/data/avrofiles/2019/1:
total 4
drwxr-xr-x 3 markayd markayd 4096 Jan 21 04:02 21

/data/avrofiles/2019/1/21:
total 4
drwxr-xr-x 2 markayd markayd 4096 Jan 21 04:20 4

/data/avrofiles/2019/1/21/4:
total 36
-rw-r--r-- 1 markayd markayd 7989 Jan 21 04:20 20.avro
-rw-r--r-- 1 markayd markayd 25314 Jan 21 04:19 15.avro
```

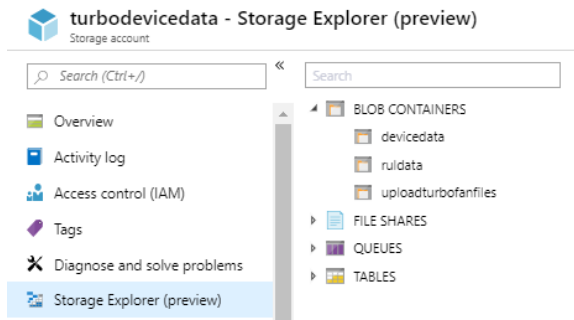
Note: you may have one or two files depending on timing of the run.

3. Pay attention to the time stamps, the avroFileWriter module will upload the files once the last modification time is more than ten minutes in the past (see MODIFIED_FILE_TIMEOUT in uploader.py in the avroFileWriter module)
4. Once the ten minutes has elapsed, the module should upload the files and if successful delete the files from disk.

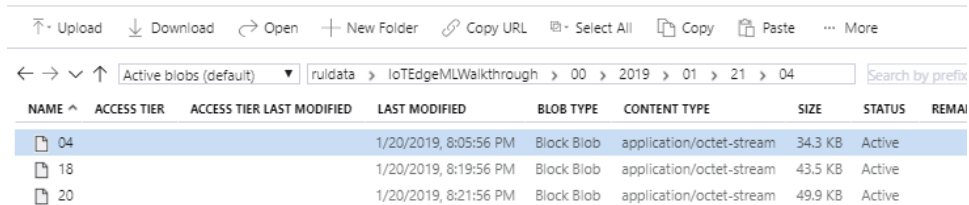
9.4.2 Azure Portal

We can observe the results of our leaf device sending data by looking at the storage accounts where we expect data to be routed.

1. Open <http://portal.azure.com> and navigate to your walk-through storage account
2. From the Storage account left navigation choose “Storage Explorer (preview)”
3. Click on “BLOB CONTAINERS” to reveal the three blob containers set up in previous steps



4. From [Reconfigure IoT Hub](#) we expect that the “ruldata” container should contain messages with RUL. Navigate down the blob hierarchy in ruldata until you reach the files.



5. Download one or more of the files to your development machine
6. Under “BLOB CONTAINERS” choose “uploadturbobanfiles”. In [Configure file upload](#), we set this location as the target for files uploaded by the avroFileWriter module.
7. Navigate through the hierarchy to the files and download one or more of these files to your development machine.

9.4.3 Read Avro file contents

We included a very simple command line utility for reading an Avro file and returning a JSON string of the messages in the file. In this section, we will install and run it.

1. Open a CMD or PowerShell window on your development machine
2. Navigate to the HubAvroReader folder:

```
cd c:\source\IoTEdgeAndMISample\HubAvroReader
```

3. Install hubavroreader

```
pip install .
```

```
c:\source\IoTEdgeAndMISample\HubAvroReader>pip install .
Processing c:\source\iotedgeandmlsample\hubavroreader
Requirement already satisfied: click in c:\python37\lib\site-packages (from hubavroreader==0.0.1) (7.0)
Requirement already satisfied: fastavro in c:\python37\lib\site-packages (from hubavroreader==0.0.1) (0.21.16)
Installing collected packages: hubavroreader
Running setup.py install for hubavroreader ... done
Successfully installed hubavroreader-0.0.1
```

4. Use hubavroreader to read an Avro file downloaded from “ruldata”

```
c:\>hubavroreader c:\AvroDownloads\04
[
  {
    "Body": {
      "ConnectionDeviceId": "aaLeafDevice_1",
      "CorrelationId": "e118a07b-dae3-4f7b-aaa1-f2c8cb0d29a1",
      "CycleTime": 1.0,
      "PredictedRul": 156.39179729188615
    },
    "EnqueuedTimeUtc": "2019-01-21T04:03:31.3070000Z",
    "Properties": {
      "AzureMLResponse": "OK",
      "ConnectionDeviceId": "aaLeafDevice_1",
      "CorrelationId": "e118a07b-dae3-4f7b-aaa1-f2c8cb0d29a1",
      "CreationTimeUtc": "01/01/0001 00:00:00",
      "EnqueuedTimeUtc": "01/01/0001 00:00:00"
    },
    "SystemProperties": {
      "connectionAuthMethod": "{\\"scope\\":\\"module\\",\\"type\\":\\"sas\\",
      "connectionDeviceGenerationId": "636835552457471566",
      "connectionDeviceId": "aaTurbofanEdgeDevice",
      "connectionModuleId": "turbofanRouter",
      "contentEncoding": "utf-8",
      "contentType": "application/json",
      "correlationId": "e118a07b-dae3-4f7b-aaa1-f2c8cb0d29a1",
      "enqueuedTime": "2019-01-21T04:03:31.3070000Z"
    }
  },
]
```

5. Note that body of the message looks as we expected with device ID and predicted RUL.

6. Run the same command passing an Avro file downloaded from “uploadturbofanfiles”

```
c:\>hubavroreader c:\AvroDownloads\20.avro
[
  {
    "Body": {
      "CycleTime": 104.0,
      "DeviceId": 87.0,
      "OperationalSetting1": 35.00790023803711,
      "OperationalSetting2": 0.8399999737739563,
      "OperationalSetting3": 100.0,
      "PredictedRul": 261.32806396484375,
      "Sensor1": 449.44000244140625,
      "Sensor10": 1.0199999809265137,
      "Sensor11": 42.09000015258789,
      "Sensor12": 182.77000427246094,
      "Sensor13": 2387.919921875,
      "Sensor14": 8066.66015625,
      "Sensor15": 9.282999992370605,
      "Sensor16": 0.01999999552965164,
      "Sensor17": 334.0,
      "Sensor18": 2223.0,
      "Sensor19": 100.0,
      "Sensor2": 555.6900024414062,
      "Sensor20": 14.979999542236328,
      "Sensor21": 8.885100364685059,
      "Sensor3": 1374.5400390625,
      "Sensor4": 1130.8199462890625,
      "Sensor5": 5.480000019073486,
      "Sensor6": 8.0,
      "Sensor7": 194.82000732421875,
      "Sensor8": 2222.889892578125,
      "Sensor9": 8355.7001953125
    },
    "ConnectionDeviceId": "aaLeafDevice_1",
    "CorrelationId": "dd15aaa4-0501-4f3f-9314-71854a12cea0",
    "CreationTimeUtc": "0001-01-01T00:00:00+00:00",
    "EnqueuedTimeUtc": "0001-01-01T00:00:00+00:00"
  },
]
```

7. As expected, these messages contain all the sensor data and operational settings from the original message. This is the data that we would use to continue to improve the RUL model on our edge device.

9.5 Summary

In this section we used our development machine to simulate a leaf device sending sensor and operational data to our edge device. We validated that the modules on the device routed, classified, persisted and uploaded the data first by examining the real time operation of the edge device and then by looking at the files uploaded to the storage account.

More information can be found at the following pages:

1. [Connect a downstream device to an Azure IoT Edge gateway](#)
2. [Store data at the edge with Azure Blob Storage on IoT Edge \(preview\)](#)