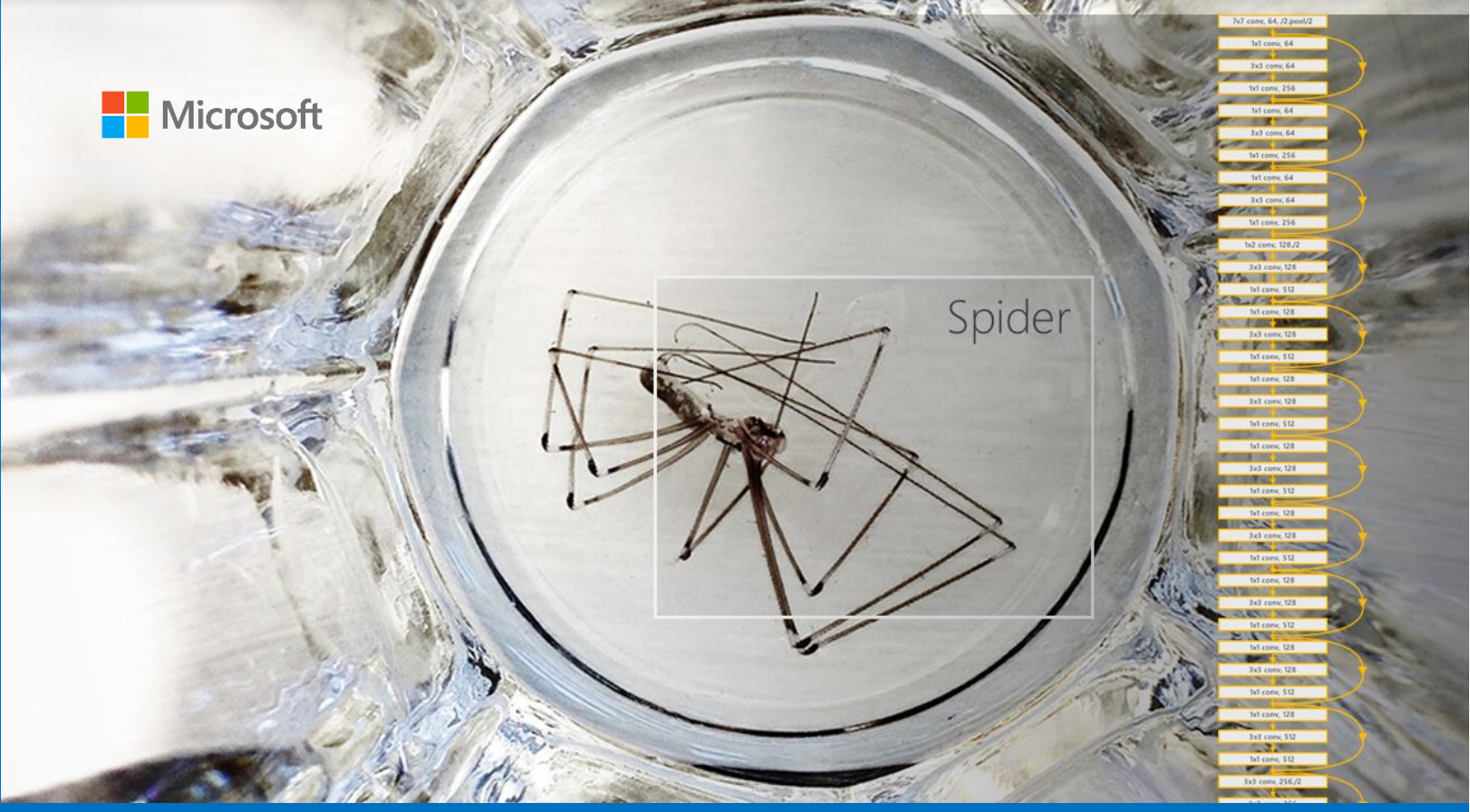


Bringing Computer Vision models to the Intelligent Edge (with Azure IoT Edge)

A guide for developers and data scientists

Version 1.1, September 2019 (Updated: November 2019)

For the latest information on the Azure IoT, please see
https://azure.microsoft.com/en-us/overview/iot/

For the latest information on the AI platform, please see
https://www.microsoft.com/en-us/ai/ai-platform

This page is intentionally left blank.

Table of contents

Notice

This guide for developers and data scientists is intended to illustrate how to build and deliver an end-to-end Computer Vision solution on edge devices with Azure IoT Edge. For that purpose, it features both the Azure IoT platform and the Microsoft AI platform.

MICROSOFT DISCLAIMS ALL WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, IN RELATION WITH THE INFORMATION CONTAINED IN THIS WHITE PAPER. The white paper is provided "AS IS" without warranty of any kind and is not to be construed as a commitment on the part of Microsoft.

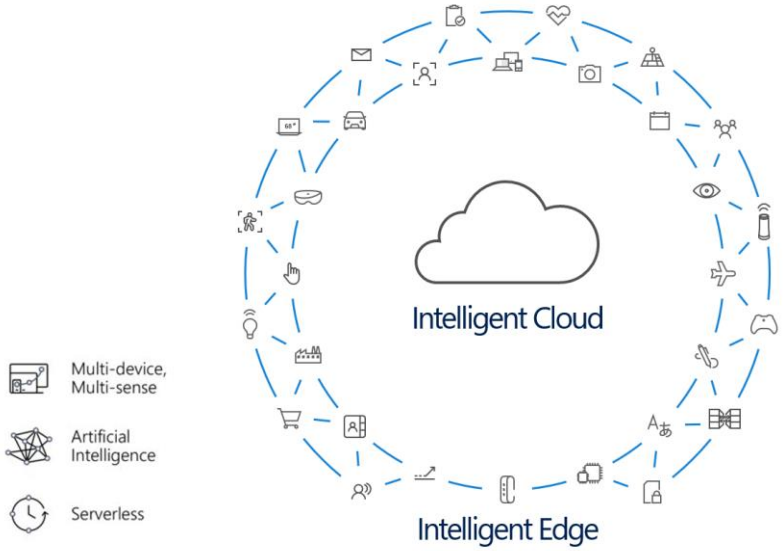
Microsoft cannot guarantee the veracity of the information presented. The information in this guide, including but not limited to internet website and URL references, is subject to change at any time without notice. Furthermore, the opinions expressed in this guide represent the current vision of Microsoft France on the issues cited at the date of publication of this guide and are subject to change at any time without notice.

Publication: September 2019 (Updated: November 2019)
Version 1.1

About this guide

Welcome to the **Bringing Computer Vision models to the Intelligent Edge** guide for both developers and data scientists.

Beside the ongoing shift for more and more organizations to the "Intelligent Cloud", i.e. an ubiquitous computing, enabled by the public cloud and Artificial Intelligence (AI) technology - Built-in AI capabilities that keeps growing[1] as illustrated as part of the last Microsoft Build 2019 conference[2]. AI is no longer an exotic concept -, for every type of intelligent application and system you can envision, we observe the inevitable rise of the "Intelligent Edge".



The "Intelligent Edge" is a continually expanding set of connected systems and devices that gather and analyze data - close to your users, the data, or both. As the (Industrial) Internet of Things ((I)IoT) indeed continues to increasingly accelerate and businesses realize the immense benefits, the next breakthrough capability upon us - or even already here as industry trends - is to enable (I)IoT edge devices *themselves* to evolve towards a new world with:

- Billions of connected multi-sense devices on the "Intelligent Edge".
- Seamless access to AI inference and insights at the Edge, powered by the "Intelligent Cloud".
- Serverless compute power leveraged at the Edge.

Enabling intelligence on edge devices means enabling analytics and insights to happen closer to the source of the data, saving organizations money and simplifying their solutions. Increased functionalities and computing power

[1] MAKING AI REAL FOR EVERY DEVELOPER AND EVERY ORGANIZATION: https://azure.microsoft.com/en-us/blog/making-ai-real-for-every-developer-and-every-organization/

[2] Microsoft Build 2019 Conference: https://news.microsoft.com/build2019/

available on the Edge are already changing the way organizations design and build products, from intelligent construction site video surveillance[3], to oil rig maintenance tracking[4].

As such, the "Intelligent Edge" hardware ecosystem includes everything from traditional PCs, servers, and mobile devices to sensors, fixed purpose devices, and microcontrollers – if it's got silicon and is Cloud-connected, it's an edge hardware. "Intelligent Edge" devices work with the "Intelligent Cloud" to seamlessly support distributing workloads like command and control (C2C), predictive maintenance, and any other AI-driven capabilities to the most appropriate node (whether it's an edge device, a gateway device, an appliance, or the latency Cloud) based on requirements around, performance, and security/compliance, etc.

We are seeing this shift across many different industries – it is not just technology companies, but companies of all types who are digitally transforming and establishing their Cloud and Edge strategies by pushing boundaries with the applications and experiences they build.

The virtually limitless computing power of the Cloud, combined with increasingly connected and perceptive devices at the Edge, create possibilities we could only have dreamed of just a few years ago – possibilities made up of millions of connected devices, infinite data, and the ability to create truly immersive multi-sense, multidevice experiences. As Julia White, Corporate Vice President, Microsoft Azure, outlines:

> ***The intelligent cloud and intelligent edge application pattern, transforms the way we can interact with digital information and further blend the physical and digital worlds for greater societal benefit and customer innovation.[5]***

Enabling "Intelligent Cloud and Intelligent Edge" solutions requires a new class of distributed, connected applications that will deliver break-through business outcomes.

As far as the "Intelligent Cloud" is concerned, this is exactly what we propose to do with the Microsoft Enterprise Cloud services, thanks to the power, the diversity and the flexibility of the services offered. While many of our competitors can only offer a segmented cloud approach and require third-party integration solutions, Microsoft has a comprehensive set of integrated cloud computing offerings, Infrastructure-as-a-Service (IaaS) to the Platform-as-a-Service (PaaS) and all its offerings from Software-as-a-Service (SaaS).

Microsoft Azure[6] represents our foundation for the "Intelligent Cloud". Microsoft Azure is a growing collection of IaaS and PaaS cloud services notably for both IoT (Azure IoT Hub[7], Azure IoT Central[8], etc.) and AI (Azure Machine Learning Service[9], Azure Databricks[10] (Spark ML), Cognitive Services[11], etc.) that allow you to move

[3] INTELLIGENT VIDEO SURVEILLANCE: AN UNTAPPED SOURCE OF VALUE: https://blog.deepomatic.com/en/intelligent-video-surveillance-an-untapped-source-of-value/4098/

[4] EDGE TECHNOLOGIES GIVE OIL, GAS OPERATORS EXTRA COMPUTING POWER: https://www.rigzone.com/news/edge_technologies_give_oil_gas_operators_extra_computing_power-30-aug-2018-156789-article/

[5] INTELLIGENT EDGE INNOVATION ACROSS DATA, IOT, AND MIXED REALITY: https://azure.microsoft.com/en-us/blog/intelligent-edge-innovation-across-data-iot-and-mixed-reality/

[6] Microsoft Azure: https://azure.microsoft.com/en-us/services/

[7] Azure IoT Hub: https://azure.microsoft.com/en-us/services/iot-hub/

[8] Azure IoT Central: https://azure.microsoft.com/en-us/services/iot-central/

[9] Azure Machine Learning Service: https://azure.microsoft.com/en-us/services/machine-learning-service/

[10] Azure DataBricks: https://azure.microsoft.com/en-us/services/databricks/

[11] Azure Cognitive Services: https://azure.microsoft.com/en-us/services/cognitive-services/

faster, achieve more, and save money. Azure serves as a development, service hosting, and service management environment, providing organizations with on-demand and pay-as-you-go (PAYG) resources, and content delivery capabilities to host, scale, and manage applications and APIs on the Internet.

With, as of this writing, 54 regions around the world, 120 000 new customers per month or 1.2 million of developers who have already worked with Cognitive Services for their modern applications or innovations like agents, bots and new conversational interfaces, Azure has established itself in recent years as a great leader in the cloud market.

While much attention has been paid to the cloud innovations, the advancements at the Edge are becoming equally remarkable. So, as far as the "Intelligent Edge" is concerned, Microsoft provides an ever-growing rich array of discreet building blocks (Azure IoT Edge[12], Azure Data Box Edge[13], Azure FPGA[14], HoloLens 2[15], Windows 10 IoT[16], Windows Machine Learning[17], etc.), and solution acceleration products that can be leveraged to build compelling end-to-end "Intelligent Cloud and Intelligent Edge" solutions.

By combining the virtually limitless computing power of the cloud with increasingly connected and perceptive devices at the edge, we're creating possibilities we could only have dreamed of just a few years ago. These possibilities apply to millions of connected IoT devices, virtually limitless data, etc.

But the most compelling solutions will be realized when it is easy for both our customers and partners to see how these many discreet building blocks can act as one end-to-end solution and combined to enable new (breakthrough) scenarios, with new revenue streams, incentives and new business partnerships.

Moreover, having all the pieces isn't enough… if putting the pieces together into end-to-end solutions is highly custom, complex and expensive – As you know, disjointed solutions require costly, bespoke development -.

The "Intelligent Cloud and Intelligent Edge" world view aspires to make complex distributed computing scenarios, which provide and/or sustain new business models, easier to discover, build, and support. This world view presupposes that the whole is greater than the sum of the parts.

[12] Azure IoT Edge: https://azure.microsoft.com/en-us/services/iot-edge/

[13] Azure Data Box Edge: https://azure.microsoft.com/en-us/services/databox/

[14] WHAT ARE FPGAS AND PROJECT BRAINWAVE?: https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-accelerate-with-fpgas

[15] HoloLens 2: https://www.microsoft.com/en-us/hololens

[16] Windows 10 IoT: https://developer.microsoft.com/en-us/windows/iot

[17] WINDOWS MACHINE LEARNING: https://docs.microsoft.com/en-us/windows/ai/windows-ml/

As outlined by Satya Nadella during an employees' Q&A session in April 2018:

> ***Building applications for multi-device, multi-sense experiences us going to require a very different form of computing architecture. That's the motivation for bringing together all of our systems people… Silicon in the edge to the silicon in the cloud architected as one workload that is distributed – that's the challenge in front of us.***[18]

And this challenge is being addressed with massive investments by Microsoft, with the end goal to give every organization the ability to transform their businesses, and the world at large, with connected solutions.

Note For additional information, see blog post MICROSOFT WILL INVEST $5 BILLION IN IOT. HERE'S WHY[19].

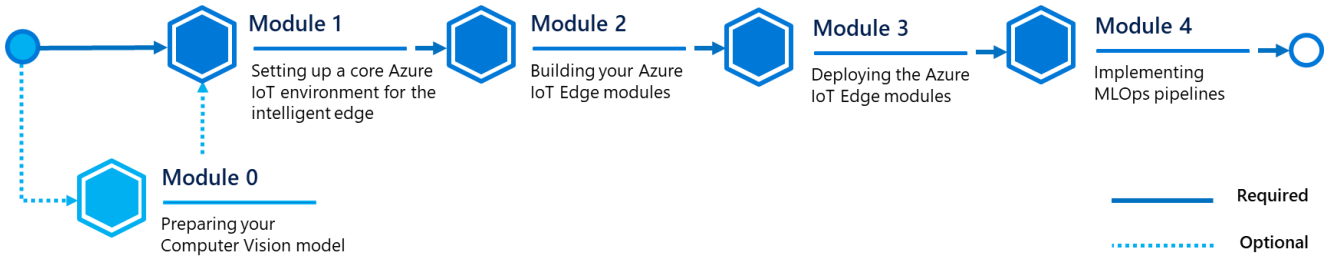
In this context, **the primary objective of this guide is to introduce you with the processing of IoT data with machine learning or deep learning models specifically on the edge.**

While you will touch many aspects of a general Machine Learning workflow for Computer Vision, this guide is not intended as an in-depth introduction to Machine Learning or Deep Learning. As a case in point, we do not attempt to create a highly optimized Deep Learning model for the use case (see section below) – we just do enough to illustrate the process of creating and using a viable Computer Vision model for IoT video data processing. The same considerations apply to the edge device being used – the device just offer enough power to run the model -.

This guide will more specifically walk you through an end-to-end AI object detection on a Raspberry Pi 3 edge device, starting (almost) "from scratch" to consider each builing blocks of such a solution. This guide is designed for developers as well as data scientists who wish to easily put their AI models in practice on edge devices without focusing too much on the deployment.

For that purposes, you're invited to follow a short series of modules, each of them illustrating a specific aspect of the IoT solution development.

Each module within the guide builds on the previous. However, since the solution is intended to be fully customizable, some modules or module contents are optional. Whenever this is the case, you will see an "(optional)" statement at the end of the title.



[19] MICROSOFT WILL INVEST $5 BILLION IN IOT. HERE'S WHY.: https://blogs.microsoft.com/iot/2018/04/04/microsoft-will-invest-5-billion-in-iot-heres-why

At the end of the guide, you will be able to:

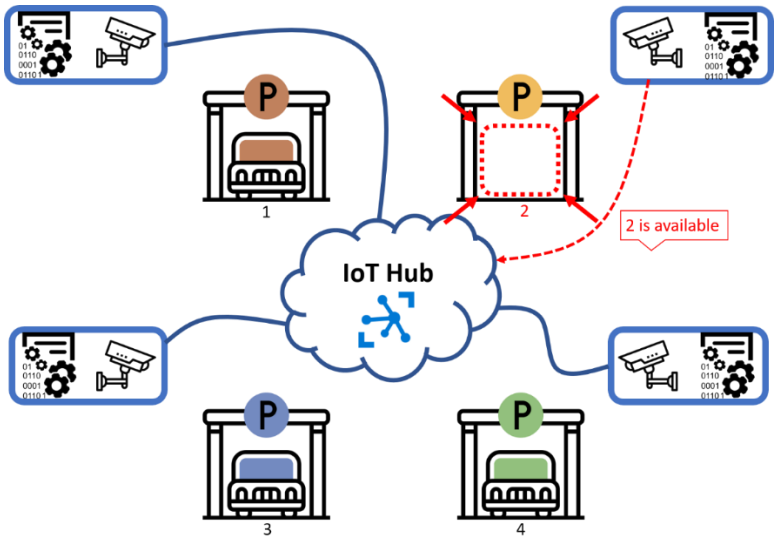
- Train a Computer Vision AI model with Azure Machine Learning from scratch (optional).
- Set up a core Azure IoT platform in the cloud, register and connect an edge device to it.
- Create custom computing modules ready to be deployed on the Azure IoT Edge device.
- Deploy and use the customs computing modules on the Azure IoT Edge device.
- Implement an MLOps pipeline with Azure DevOps[20].

Guide use case

Imagine you are the head of a parking lot and you want to easily manage available parking spaces without checking by yourself or without investing in expensive sensors or external services.

What if you could automate everything using only the surveillance cameras you bought years ago? Nowadays, there is an increasing flow of datas that are collected through edge devices, whether it is to measure the temperature of the room or to monitor surveillance at home. The Internet of Things (IoT) has brought us the question of how to manage all of this efficiently and in a secure way. For the past few years, as outline dbefore, Azure cloud services have made astounding progress for the IoT: with the introduction of Azure IoT Edge back in 2018, you can now move workloads on edge devices at will, monitor in-and-out flow of datas remotely and with ease.

Back to our parking lot, the objective here is to run a custom Computer Vision model that can detect all types of car on the surveillance cameras. However, we don't want to monitor it by ourselves, available spaces should only be reported if a spot has been freed and not every time there is a movement in the parking lot.



[20] Azure DevOps: https://azure.microsoft.com/en-us/services/devops/

This is what makes Azure IoT Edge the perfect match: all the computations are given to the edge devices, resulting in less interactions with the cloud and you would be able to monitor your parking lot remotely on Azure IoT Hub. You can make edge devices do all the heavy work for you: for instance, with an object detection, it can tell you if the car is parked correctly, if there is a person on the spot, etc.

Guide elements

In most guide's modules, you will see the following elements:

- **Step-by-step directions**. Click-through instructions - along with relevant snapshots - or links to online documentation for completing each procedure.
- **Important concepts**. An explanation of some of the concepts important to the procedures in the module, and/or what happens behind the scenes.
- **Sample applications, and files**. A downloadable version of the IoT solution that you will use in this guide, and other files, such as Jupyter notebook files, you will need as well as part of the wakkthrough. **Please go to https://aka.ms/IEdgeDevGuideSamples to download all necessary assets.**
- **Highly commented code snippets.** The various code snippets throughout this guide are intended to illustrate how applications are created and how to re-use and customize the provided codes, scripts and configuration files to design your own tailored for purpose IoT solution. **Please pay attention on the comments as you will not go line by line each code.**

Guide prerequisites

To successfully set up this solution in this guide, you will need:

- A Microsoft account[21].
- An Azure subscription. If you don't have an Azure subscription, create a free account[22] before you begin.
- The project which contains the code, scripts and configuration files in any for every module, and related activites, of this guide, you can download it or clone it here[23] from GitHub.

Optionally, you may need additional bits for each module, and the related step-by-step directions, in this guide. Check out the tab below if you want to know in advance the requirements needed for each module.

Note	Some requirements can be redundant depending of parts choosed.

[21] Microsoft Account: https://account.microsoft.com/account?lang=en-us

[22] Create your Azure free account today: https://azure.microsoft.com/en-us/free/?WT.mc_id=A261C142F

[23] Guide repo: https://aka.ms/IEdgeDevGuideSamples

Module/Part	Requirements
Prepare Computer Vision model	• Python environment (minimum Python 3.5+) • Azure Notebook or Jupyter Notebook
Set up the Azure IoT Hub > Set up the Raspberry Pi	• A physical Raspberry Pi 3 Model B • Raspbian Stretch
Build the Computer Vision module > Create container image	• Vistual Studio Code • Azure IoT Edge Visual Studio Code extension • Docker CE
Set up Camera Module > Connect the Pi Camera	• A physical Raspberry Pi Camera V2
Set up Camera Module > Deployment	• Visual Studio Code • Azure IoT Edge Visual Studio Code extension • Docker CE
MLOps	• 'Owner' or 'User Access Administrator' permissions on the Workspace • Sufficient permissions to create service principals on Azure Active Directory

Guide audience and roles

This guide is intended for developers without previous experience in IoT development, and/or Machine Learning/Deep Learning. Deploying Machine Learning at the edge requires knowledge of how to connect a wide range of technologies. Therefore, this walkthrough covers an entire end-to-end scenario (see above section) to demonstrate one way of joining these technologies together for an IoT solution. In a real-world environment, these tasks might be distributed among several people with different specializations.

For example, developers would focus on either (edge) device or cloud code, while data scientists designed the AI models. This is the reason why this guide is targeted for both population: developers and data scientists.

So, to enable an individual developer/data scientist to successfully complete this walkthrough and its several modules, we have provided supplemental guidance with insights and links to more information we hope is sufficient to understand what is being done, as well as why.

Alternatively, you may team up with coworkers of different roles to follow the walkthrough together, bringing your full expertise to bear, and learn as a team how things fit together.

In either case, to help orient the reader(s), each activities in this guide indicates the role of the user. Those roles include:

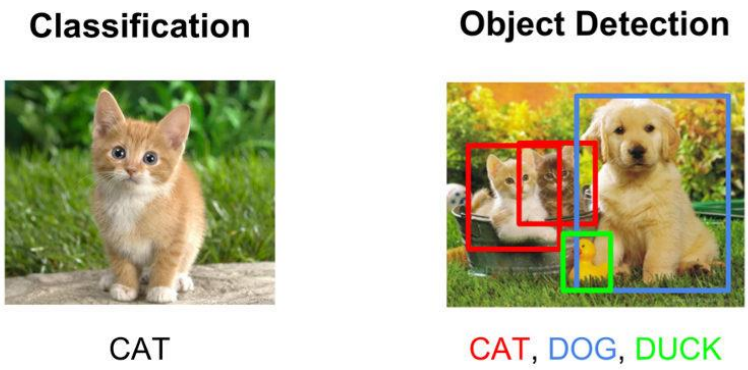
- Cloud developer,
- Device developer,
- Data scientists,
- DevOps engineers.

Module 0: Preparing your Computer Vision model (optional)

Overview

Depending on what task you are facing, you need to choose an AI model that would suit your needs. In fact, if all that matters to you is to classify a given object, Deep Learning convolutional networks might be the way to go. However, you might be tempted to also have the position of a classified object, in this case, you may need to consider another neural networks. Here are few main Computer Vision problems:

- Image classification,
- Object detection,
- Object segmentation,
- …



Hence, depending on the problem, determining the right AI model is crucial for the rest. For the purposes of this guide, you are going to use the state-of-the-art real-time object detection YOLO v3[24], i.e. the latest version of You Only Look Once[25] (YOLO) - a wink to the aphorism "You Only Live Once" -.

Among object detection neural networks, although it is a little less accurate than others, the trade-off with the speed is huge, which makes it a good model to deploy on edge devices.

[24] WHAT'S NEW IN YOLO V3?: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b

[25] YOLO: https://pjreddie.com/darknet/yolo/

Model	Train	Test	mAP	FLOPS	FPS
SSD300	COCO trainval	test-dev	41.2	-	46
SSD500	COCO trainval	test-dev	46.5	-	19
YOLOv2 608x608	COCO trainval	test-dev	48.1	62.94 Bn	40
Tiny YOLO	COCO trainval	test-dev	23.7	5.41 Bn	244
SSD321	COCO trainval	test-dev	45.4	-	16
DSSD321	COCO trainval	test-dev	46.1	-	12
R-FCN	COCO trainval	test-dev	51.9	-	12
SSD513	COCO trainval	test-dev	50.4	-	8
DSSD513	COCO trainval	test-dev	53.3	-	6
FPN FRCN	COCO trainval	test-dev	59.1	-	6
Retinanet-50-500	COCO trainval	test-dev	50.9	-	14
Retinanet-101-500	COCO trainval	test-dev	53.1	-	11
Retinanet-101-800	COCO trainval	test-dev	57.5	-	5
YOLOv3-320	COCO trainval	test-dev	51.5	38.97 Bn	45
YOLOv3-416	COCO trainval	test-dev	55.3	65.86 Bn	35
YOLOv3-608	COCO trainval	test-dev	57.9	140.69 Bn	20
YOLOv3-tiny	COCO trainval	test-dev	33.1	5.56 Bn	220
YOLOv3-spp	COCO trainval	test-dev	60.6	141.45 Bn	20

The performances above have been obtained by comparing different neural networks on the same Common Objects in Context[26] (COCO) dataset, a large-scale object detection, segmentation, and captioning dataset, which contains more than 80 classes.

Note For more information, see whitepaper MICROSOFT COCO: COMMON OBJECTS IN CONTEXT[27].

The tiny versions of YOLO can reach up to 200+ frame per second (FPS), which can be run in real-time even on resource-constrained device such as ARM32 devices.

Training can be challenging since it is highly dependent on the hardware you train your neural network on. Some can take multiple hours to complete or even days/weeks. Furthermore, a simple power failure is all it takes to ruin your training. You may have saved checkpoints but you need to train it again, and most of the time, you are not there watching it train. You may also not have the hardware powerful enough to train your neural network in reasonable time.

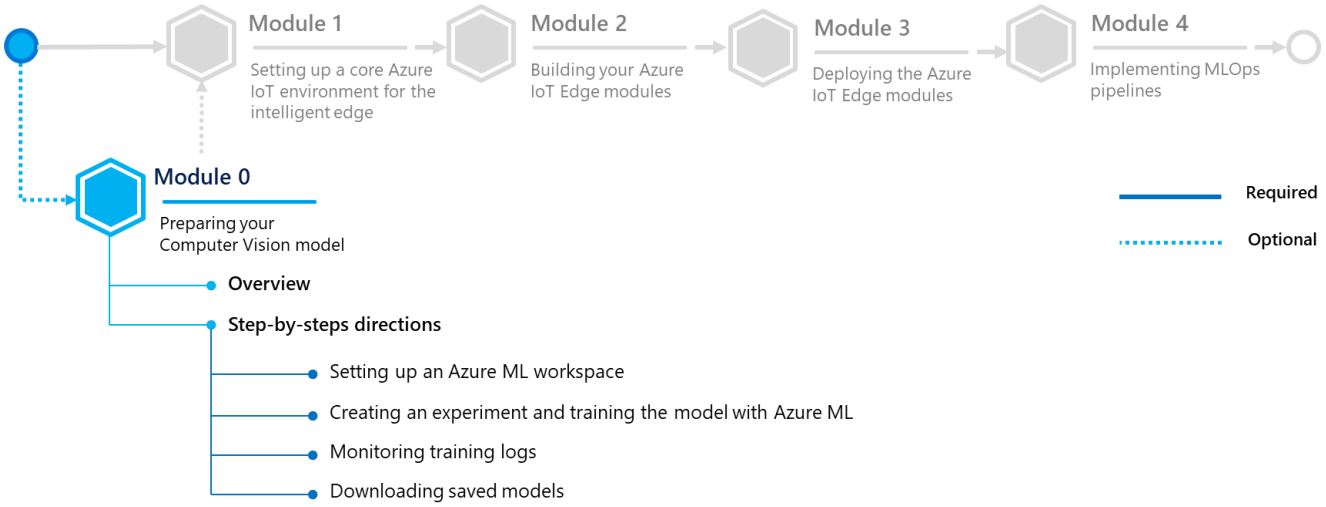
Azure Machine Learning (Azure ML) Service enables you to train your neural network on a remote machine you can choose from a wide range of options, meaning you don't have to worry about the execution nor the uptime of the hardware. Moreover, the cost will smoothly adjust to the execution time so that you will only pay for what it takes for the training. On the Azure portal, you can also follow the training of your neural network so that you can visualize the metrics and logs remotely and as the training goes.

In this module, you will learn to:

- Train an object detection model with Azure ML and a Jupyter Notebook file.
- Save your model.

[26] COCO: http://cocodataset.org/#home

[27] MICROSOFT COCO: COMMON OBJECTS IN CONTEXT: https://arxiv.org/abs/1405.0312



Note If you are only interested in having a custom image classifier/object detection ready to be used, Microsoft has recently released the Custom Vision[28] service as part of the Cognitive Services[29] that enables you to customize state-of-the-art computer vision models with ease: you just need to bring a few examples of labeled images and let Custom Vision do the hard work.

For more information, see tutorials QUICKSTART: HOW TO BUILD A CLASSIFIER WITH CUSTOM VISION[30] and QUICKSTART: HOW TO BUILD AN OBJECT DETECTOR WITH CUSTOM VISION[31] - Make sure to improve your model[32] if needed -.

If you already have a suitable Computer Vision model, you may skip this module.

Step-by-step directions

This module covers the following activities:

1. Creating an Azure ML experiment and training the model with Azure ML,
2. Visualizing training logs,
3. Downloading saved models.

Each activity is described in order in the next sections.

You can follow the whole process by running the notebook *00-train-yolov3.ipynb* available here[33].

---

[28] Custom Vision: https://www.customvision.ai/

[29] Cognitive Services: https://azure.microsoft.com/en-us/services/cognitive-services/

[30] QUICKSTART: HOW TO BUILD A CLASSIFIER WITH CUSTOM VISION: https://docs.microsoft.com/en-us/azure/cognitive-services/custom-vision-service/getting-started-build-a-classifier

[31] QUICKSTART: HOW TO BUILD AN OBJECT DETECTOR WITH CUSTOM VISION: https://docs.microsoft.com/en-us/azure/cognitive-services/custom-vision-service/get-started-build-detector

[32] HOW TO IMPROVE YOUR CLASSIFIER: https://docs.microsoft.com/en-us/azure/cognitive-services/custom-vision-service/getting-started-improving-your-classifier

[33] 00-train-yolov3.ipynb: https://aka.ms/IEdgeDevGuideSamples

We recommend to use for that purpose Azure Notebooks[34]. Azure Notebooks take advantage of an Azure Machine Learning service workspace, which is a foundational block used to experiment, train, and deploy machine learning models. You can alternatively use Jupyter Notebook[35] to view and run the scripts of the above notebook.

Note If you have not used Jupyter and/or Azure Notebooks, here are a couple of introductory documents: QUICKSTART: CREATE AND SHARE A NOTEBOOK[36] and TUTORIAL: CREATE AND RUN A JUPYTER NOTEBOOK WITH PYTHON[37].

Using Azure Notebooks ensures a consistent environment for the module.

To set up Azure Notebook, see section § **Setting up Azure Notebooks** in the Appendix. Using Azure Notebooks. Once set up, the Azure Notebooks service can be accessed from any machine.

Setting up an Azure ML workspace

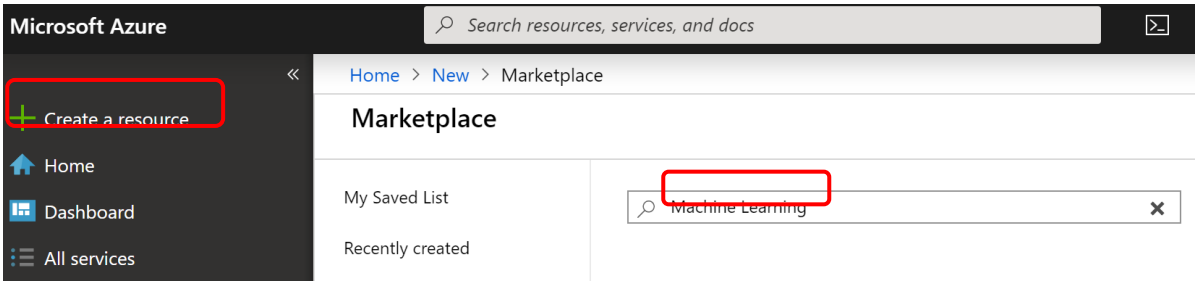
The steps in this section might be typically performed by data scientists.

You can either manually create an Azure ML workspace on Azure portal or use az commands of Azure CLI[38]. (Azure CLI is a (cross-platform) command-line tool providing a great experience for managing Azure resources. The CLI is designed to make scripting easy, query data, support long-running operations, and more.)

If you are following the notebook, you will need to install the **Azure CLI** - you can follow the instructions here[39] -.

To manually create a workspace, perform the following steps:

1. Sign in to the Azure portal at https://portal.azure.com.
2. In the left pane, select **Create a resource**. Search for "*Machine Learning*" in the **Search the Marketplace** search bar.



3. Select **Machine Learning service workspace**.

[34] Azure Notebooks: https://notebooks.azure.com/

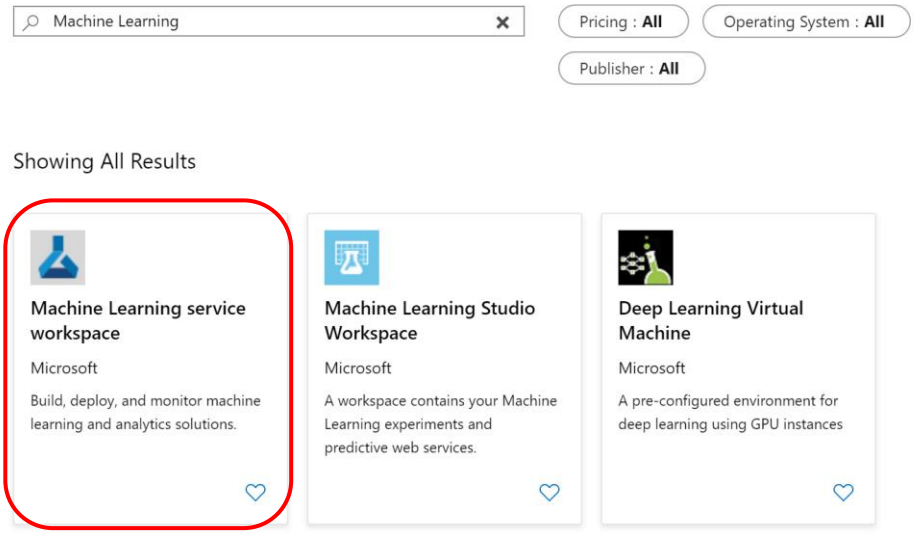
[35] Jupyter Notebook: https://jupyter.org/

[36] QUICKSTART: CREATE AND SHARE A NOTEBOOK: https://docs.microsoft.com/en-us/azure/notebooks/quickstart-create-share-jupyter-notebook

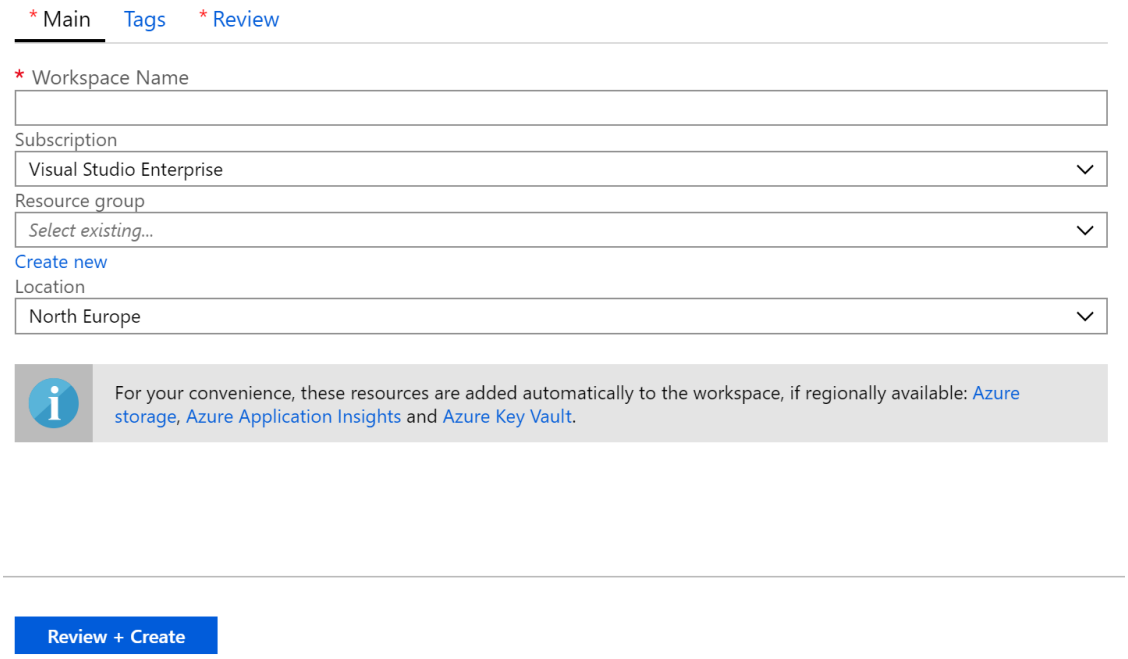
[37] TUTORIAL: CREATE AND RUN A JUPYTER NOTEBOOK WITH PYTHON: https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-machine-learning-edge-04-train-model

[38] THE AZURE COMMAND-LINE INTERFACE (CLI): https://docs.microsoft.com/en-us/cli/azure/?view=azure-cli-latest

[39] INSTALL THE AZURE CLI: https://docs.microsoft.com/en-us/cli/azure/install-azure-cli?view=azure-cli-latest

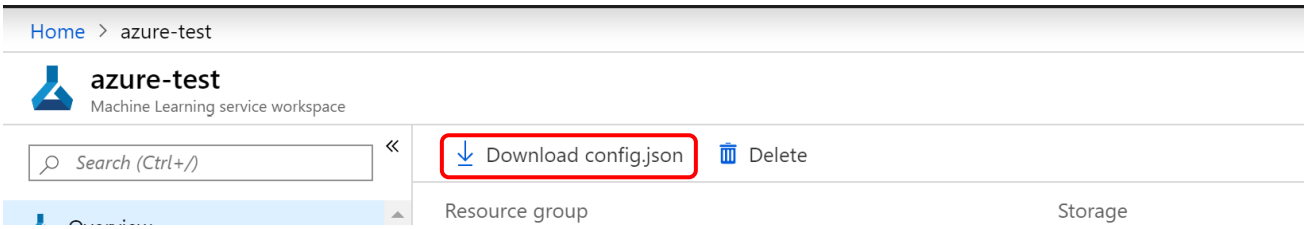


4. Select **Create**.
5. Complete the settings and select **Review + Create.**



Setting	Description
Workspace Name	Select a name for your workspace.
Subscription	Specify the Azure subscription you wish to use for your deployment.
Resource groups	Create a new resource group by selecting **Create new** and specify a unique resource group name.
Location	Specify the region in which you want to deploy the framework.

6. Select **Create** and your workspace is ready to go. Once the workspace is live, you can get the *config.json* file from the Azure portal:



Copy this file into the same directory as your notebook, you will need it for the next step.

Creating an Azure ML experiment and training the model with Azure ML

The steps in this section might be typically performed by data scientists.

Each training must be registered under an Azure ML experiment. Open up the notebook *00-train-yolov3.ipynb*.

From what you did above, you should already have a *config.json* file containing all the details of your azure account. You can skip the **requirements section** if you have the *config.json* file. (Otherwise, you can create an azure ML workspace by running the commands in the first section).

To initialize your Azure ML workspace, you just have to run:

```
from azureml.core import Workspace
import azureml.core

ws = Workspace.from_config()
```

To create an experiment, simply run:

```
from azureml.core import Experiment
import azureml.core

exp = Experiment(workspace=ws, name='yolov3')
```

Uploading and accessing training data

Data is something that you must be really careful about whether they come from customers or a private database. Interestingly enough, Azure ML provide datastore to upload and mount your database at will. Once registered, you won't need to provide any secret key and your data won't be duplicated - You can access it in any Azure ML computing environment -. Thus, storing your data in a datastore is really crucial for training without leaking any secret key in your source code, hence, protecting them.

You can register your own datastore:

```
from azureml.core import Datastore

# register a new Azure Blob container datastore
datastore = Datastore.register_azure_blob_container(workspace=ws,
                                                    datastore_name='YOUR_DATASTORE_NAME',
```

```
                                                    container_name='YOUR_CONTAINER_NAME',
                                                    account_name='YOUR_ACCOUNT_NAME',
                                                    account_key='YOUR_STORAGE_ACCOUNT_KEY',
                                                    create_if_not_exists=True)

# register a new Azure Datastore
datastore = Datastore.register_azure_file_share(workspace=ws,
                                                datastore_name='YOUR_DATASTORE_NAME',
                                                container_name='YOUR_CONTAINER_NAME',
                                                account_name='YOUR_ACCOUNT_NAME',
                                                account_key='YOUR_STORAGE_ACCOUNT_KEY',
                                                create_if_not_exists=True)

# get your own registered datastore
ds = Datastore.get(ws, datastore_name='DATASTORE_NAME')
```

Note You can get your Azure Storage access key from Azure portal or by running: `az storage account keys list -g <MyResourceGroup> -n <MyStorageAccount>` in your terminal command prompt with Azure CLI installed.

Or, you can alternatively use the default datastore that comes with your AML workspace:

```
ds = ws.get_default_datastore()
```

Either way, to upload your files into your datastore, run:

```
ds.upload(
    src_dir='SOURCE_DIR',
    target_path='TARGET_PATH',
    overwrite=True)
```

Training the model with Azure ML

For the purposes of this guide, you will be fine-tuning (transfer learning) a pre-trained YOLOv3 model on the Pascal Visual Object Classes[40] (VOC) dataset - The Pascal VOC challenge[41] is a very popular dataset for building and evaluating algorithms for image classification, object detection, and segmentation -.

For that, you will be inspired by this Keras implementation of YOLOv3 available here[42]. For the following, you need to open up the Jupyter notebook.

[40] Pascal VOC Dataset Mirror: https://pjreddie.com/projects/pascal-voc-dataset-mirror/

[41] The PASCAL Visual Object Classes Homepage: http://host.robots.ox.ac.uk/pascal/VOC/index.html

[42] A Keras implementation of YOLOv3 (Tensorflow backend): https://github.com/qqwweee/keras-yolo3

In order to train, you must choose the remote machine you plan to train your model on. For that, you have to create a cluster and attach a compute target - Here, the choice is the 'STANDARD_NC6' VM; you can find a non-exhaustive list of available VM here[43]- to it:

```
from azureml.core.compute import ComputeTarget, AmlCompute
from azureml.core.compute_target import ComputeTargetException

# choose a name for your cluster
cluster_name = "yolo-cluster"

try:
    compute_target = ComputeTarget(workspace=ws, name=cluster_name)
    print('Found existing compute target')
except ComputeTargetException:
    print('Creating a new compute target...')
    compute_config = AmlCompute.provisioning_configuration(vm_size='STANDARD_NC6',
                                                           max_nodes=1)
    # choose the compute target according to your needs
    # create the cluster
    compute_target = ComputeTarget.create(ws, cluster_name, compute_config)

    # can poll for a minimum number of nodes and for a specific timeout.
    # if no min node count is provided it uses the scale settings for the cluster
    compute_target.wait_for_completion(show_output=True, min_node_count=None, timeout_in_minutes=20)

# use get_status() to get a detailed status for the current cluster.
print(compute_target.get_status().serialize())
```

`max_nodes` can be configured to modify the size of your cluster (to decide if others can use nodes of your cluster while you train your model).

To run a training, you have to set up the Python environment (which means specifying the dependencies), the parameters for the training script, and the script itself.

As you are training with Keras, you can use the Tensorflow estimator from `azureml.train.dnn` (which contains among other things a Docker image with all the python requirements to run Tensorflow):

```
from azureml.train.dnn import TensorFlow

est = TensorFlow(source_directory='./script',
                 script_params={'--data_folder': ds.path('VOCdevkit').as_mount(), '--
model': ds.path('model_data').as_mount()},
                 compute_target=compute_target,
                 pip_packages=['keras', 'pillow', 'matplotlib'],
                 entry_script='train.py',
                 use_gpu=True)
```

Above is an example of an estimator object. You can give as many parameters as you want to the `entry_script` by feeding a dictionary to `script_params` (and then parse the informations with `argparse`). See *train.py* for more

[43] SIZES FOR WINDOWS VIRTUAL MACHINES IN AZURE: https://docs.microsoft.com/en-us/azure/virtual-machines/windows/sizes

details about how to specify custom training parameters such as the learning rate, batch size, number of epochs, etc.).

By default, the `source_directory` on Azure ML is limited in size, so you will only be able to upload scripts or files that are necessary to the training and not the data or heavy pre-trained models as in your case, which is why you uploaded them on the datastore. `.as_mount()` indicates that you mount the folders uploaded on your datastore

To see an exhaustive list of the estimator, see here[44].

Finally, to run the training:

```
run = exp.submit(est)
```

Visualizing training logs

The steps in this section might be typically performed by data scientists.

While your neural network is being trained, Azure ML allows you to log the progress on the Azure portal to visualize logs and metrics remotely on the cloud: just like Tensorboard for example, but you can visualize the results with your favorite framework.

For that, you need to modify and add few lines of code in your original script as follows:

```
# imports …

# Logging for azure ML
from azureml.core.run import Run
# Get run when running in remote
if 'run' not in locals():
    run = Run.get_context()

def _main(model_path, fine_tune_epochs, unfrozen_epochs, learning_rate):
    annotation_path = 'train.txt'
    log_dir = 'logs/000/'
    classes_path = model_path+'/voc_classes.txt'
    anchors_path = model_path+'/yolo_anchors.txt'
    class_names = get_classes(classes_path)
    num_classes = len(class_names)
    anchors = get_anchors(anchors_path)

    input_shape = (416,416) # multiple of 32, hw

    is_tiny_version = len(anchors)==6 # default setting
    if is_tiny_version:
        model = create_tiny_model(input_shape, anchors, num_classes,
            freeze_body=2, weights_path=model_path+'/tiny_yolo_weights.h5')
    else:
        model = create_model(input_shape, anchors, num_classes,
```

[44] TENSORFLOW CLASS: https://docs.microsoft.com/en-us/python/api/azureml-train-core/azureml.train.dnn.tensorflow?view=azure-ml-py

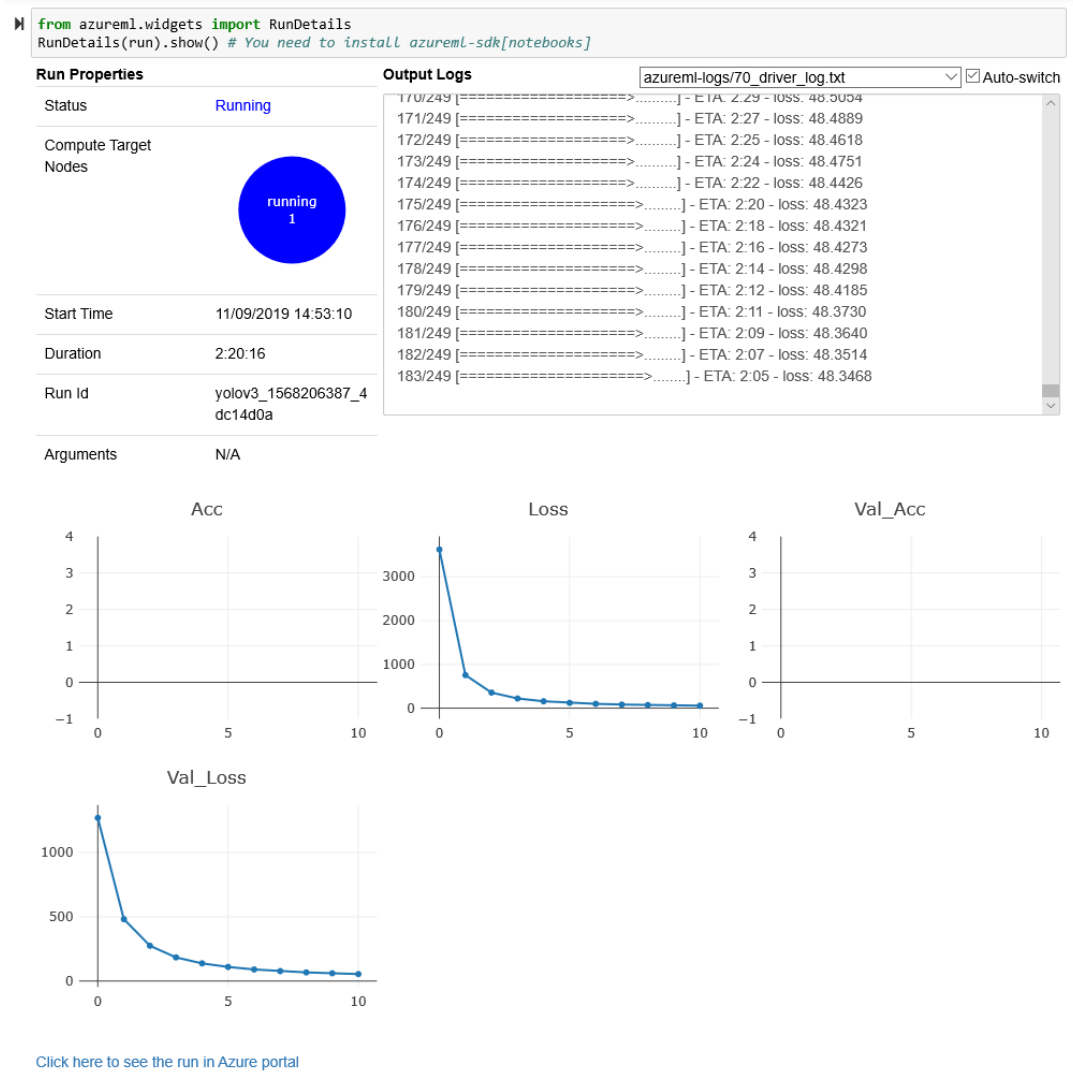
```
            freeze_body=2, weights_path=model_path+'/yolo_weights.h5') # make sure you know what you
freeze

    # Define callbacks during training
    logging = TensorBoard(log_dir=log_dir)
    checkpoint = ModelCheckpoint(log_dir + 'ep{epoch:03d}-loss{loss:.3f}-val_loss{val_loss:.3f}.h5',
        monitor='val_loss', save_weights_only=True, save_best_only=True, period=3)
    reduce_lr = ReduceLROnPlateau(monitor='val_loss', factor=0.1, patience=3, verbose=1)
    early_stopping = EarlyStopping(monitor='val_loss', min_delta=0, patience=10, verbose=1)

    # Logging for Azure ML (send acc, loss, val_loss at the end of each epoch)
    class LossHistory(Callback):
        def on_epoch_end(self, epoch, logs={}):
            run.log('Acc', logs.get('acc')) # log accuracy
            run.log('Val_Acc', logs.get('val_acc')) # log validation accuracy
            run.log('Loss', logs.get('loss')) # log loss
            run.log('Val_Loss', logs.get('val_loss')) # log validation loss

    lossHistory = LossHistory()

    val_split = 0.2
    with open(annotation_path) as f:
        lines = f.readlines()
    np.random.seed(10101)
    np.random.shuffle(lines)
    np.random.seed(None)
    num_val = int(len(lines)*val_split)
    num_train = len(lines) - num_val

    # Train with frozen layers first, to get a stable loss.
    # Adjust num epochs to your dataset. This step is enough to obtain a not bad model.
    if True:
        model.compile(optimizer=Adam(lr=learning_rate), loss={
            # use custom yolo_loss Lambda layer.
            'yolo_loss': lambda y_true, y_pred: y_pred})

        batch_size = 32
        print('Train on {} samples, val on {} samples, with batch size {}.'.format(num_train, num_val
, batch_size))
        model.fit_generator(data_generator_wrapper(lines[:num_train], batch_size, input_shape, anchor
s, num_classes),
                steps_per_epoch=max(1, num_train//batch_size),
                validation_data=data_generator_wrapper(lines[num_train:], batch_size, input_shape, an
chors, num_classes),
                validation_steps=max(1, num_val//batch_size),
                epochs=fine_tune_epochs,
                initial_epoch=0,
                callbacks=[logging, checkpoint, lossHistory])
        model.save_weights(log_dir + 'trained_weights_stage_1.h5')

    # Unfreeze and continue training, to fine-tune.
    # Train longer if the result is not good.
    if True:
        for i in range(len(model.layers)):
            model.layers[i].trainable = True
        model.compile(optimizer=Adam(lr=learning_rate), loss={'yolo_loss': lambda y_true, y_pred: y_p
red}) # recompile to apply the change
        print('Unfreeze all of the layers.')

        batch_size = 32 # note that more GPU memory is required after unfreezing the body
```

```
        print('Train on {} samples, val on {} samples, with batch size {}.'.format(num_train, num_val
, batch_size))
        model.fit_generator(data_generator_wrapper(lines[:num_train], batch_size, input_shape, anchor
s, num_classes),
            steps_per_epoch=max(1, num_train//batch_size),
            validation_data=data_generator_wrapper(lines[num_train:], batch_size, input_shape, anchor
s, num_classes),
            validation_steps=max(1, num_val//batch_size),
            epochs=unfrozen_epochs,
            initial_epoch=fine_tune_epochs,
            callbacks=[logging, checkpoint, reduce_lr, early_stopping, lossHistory])
        model.save_weights(log_dir + 'trained_weights_final.h5')

    # Further training if needed.

    ### MORE FUNCTIONS AND MAIN DEFINITION ###
```

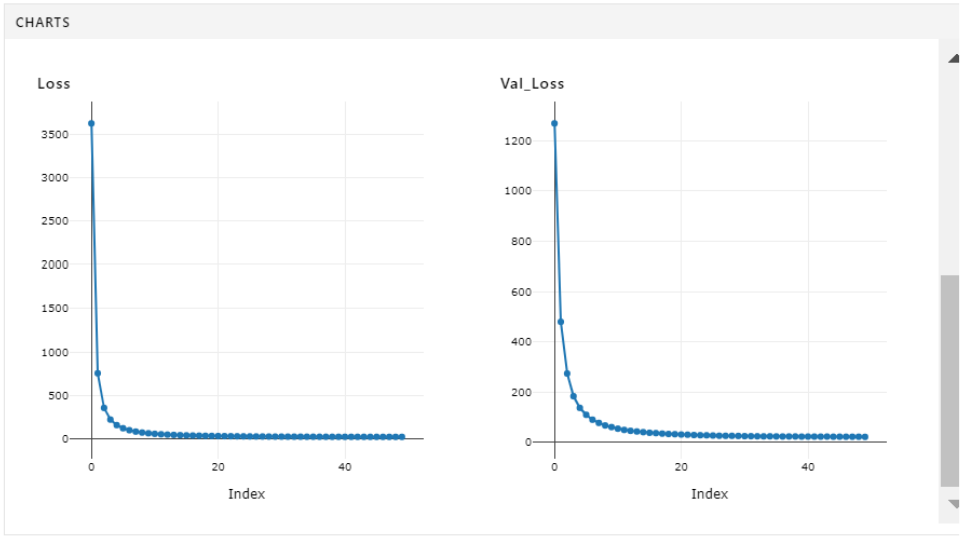
Keras allows you to define callbacks during the training, meaning that you can report logs as the model is impoving and plotting loss and accuracy graphs to follow its development. (See more here[45].)

You can either see the progress directly in your notebook as follows (if it raises an error, it is probably because you need the `azureml-sdk[notebooks]`, if you are running the notebook on Azure ML, it should not raise any error):

[45] USAGE OF CALLBACKS: https://keras.io/callbacks/



or on the Azure portal (in the Azure ML section) :

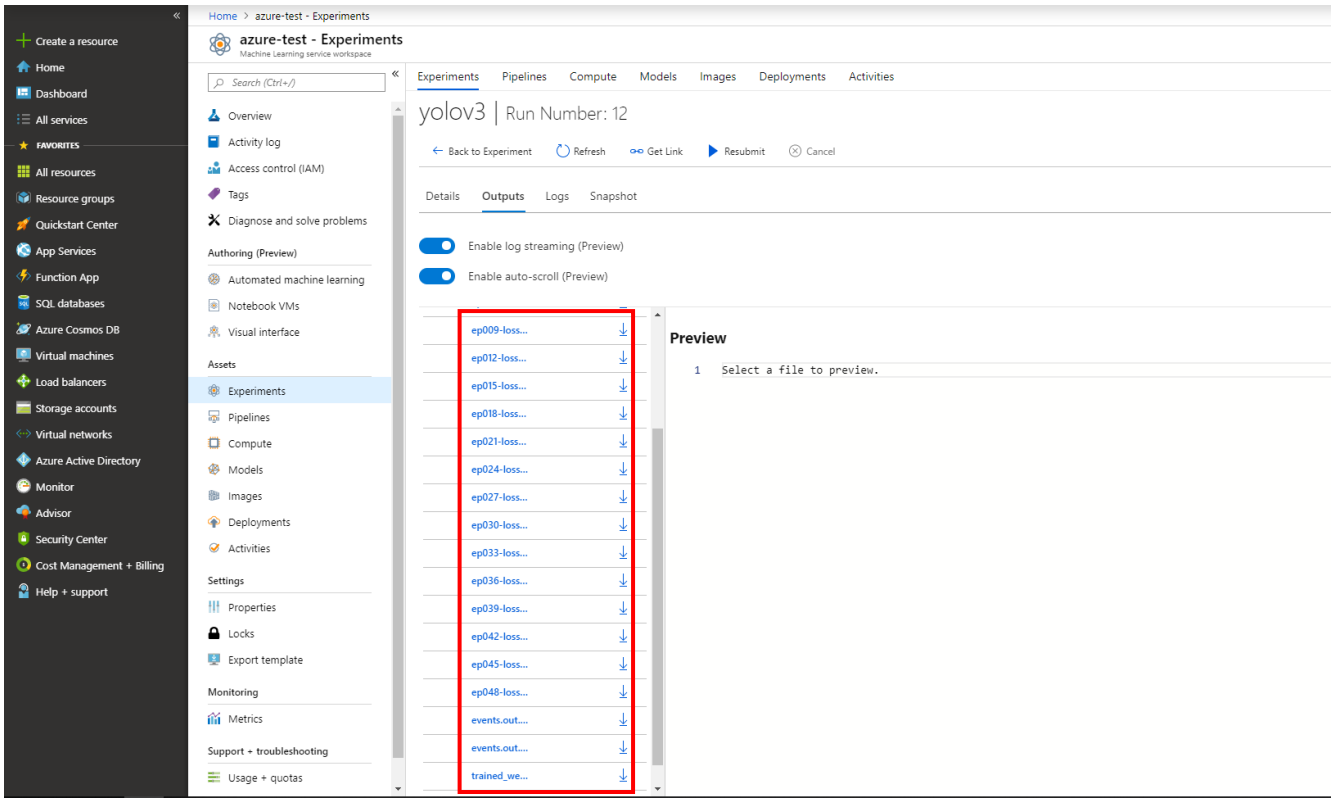


Downloading saved models

The steps in this section might be typically performed by data scientists.

Once the training is complete, you can retrieve the (checkpoints or final) models on the Azure portal.

To retrieve on the Azure portal, sign in to your account and go to the Azure ML section (Experiments > yolov3):



Module 1: Setting up a core Azure IoT environment for the intelligent edge

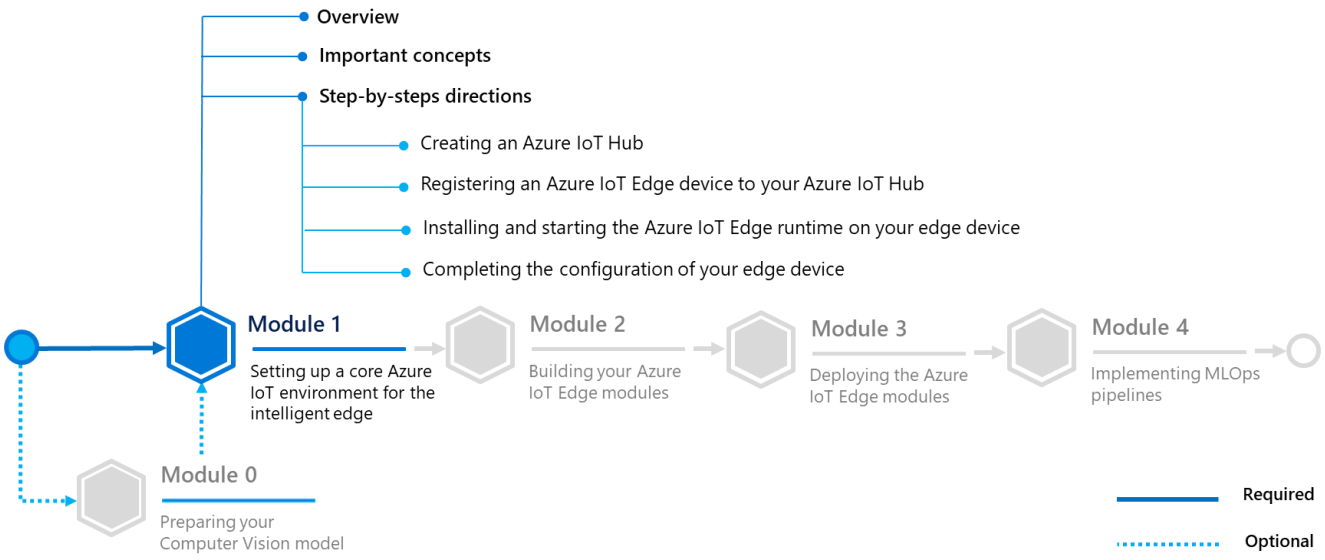
Overview

The goal in this module is to go through the process of setting up an Azure IoT[46] platform and connecting an edge device to it. As we mentioned earlier, edge computing is taking over cloud computing when it comes to managing edge devices. In fact, here are some key advantages :

- **Low-latency access**. Edge computing exposes compute, storage, and networking locally.
- **Reduced bandwidth consumption**. Edge layer aggregates and filters data by only ingesting what's needed to the public cloud.
- **Offline availability**. Applications that have intermittent access to the Internet and cloud can rely on local resources exposed by the edge computing layer.
- **Local ML inference**. Machine learning models that are deployed at the edge inference faster.

In this module, you will learn to:

- Set up an Azure IoT Hub, a core service of the Azure IoT platform,
- Register, configure and connect an edge device to your Azure IoT Hub.



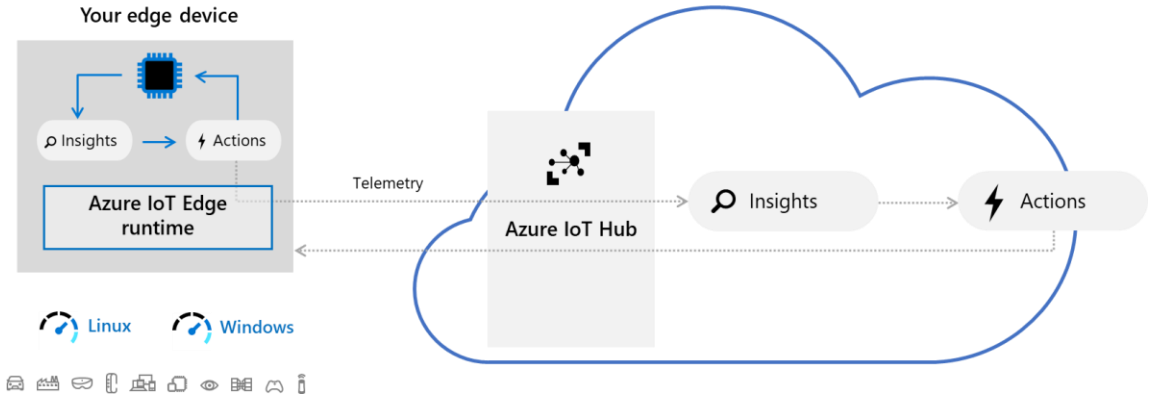
[46] Azure IoT : https://azure.microsoft.com/en-us/overview/iot/

Before diving in, let's consider further the Azure IoT platform for the so-called "Intelligent Cloud, Intelligent Edge".

Important concepts

Azure IoT Platform for the "Intelligent Cloud, Intelligent Cloud"

Generally speaking, a cloud gateway represents a key component for the so-called "Intelligent Cloud, Intelligent Edge". It indeed provides a cloud hub for (edge) devices to connect securely to the cloud and send data. It also provides device management, capabilities, including command and control of devices.



In the Azure IoT platform, Azure IoT Hub is a hosted cloud service that ingests events from (edge) devices, acting as a message broker between these devices and backend services. IoT Hub provides secure connectivity, event ingestion, bidirectional communication, and device management.

Mobile and (Industrial) Internet of Things ((I)IoT) devices can thus securely register with the cloud, here the Azure cloud, and can connect to the cloud to send and receive data. Some devices may be edge devices that perform some data processing on the device itself or in a field gateway.

The Intelligent Edge indeed brings the power of the cloud to edge devices and demands security for trust. "Cloud-enabled computing at the edge means concentrating data, and therefore inherent value even if only momentarily. It also means moving tremendous value from the cloud to the edge in the form of intellectual property, algorithms, curated parameters, and value operations like policy enforcements, metering, and monetization. The Intelligent Edge is without a doubt a high-value bullseye to nefarious hacking and demands a high bar for security."[47]

For edge processing, we then recommend Azure IoT Edge. Azure IoT Edge is an implementation of a secure Intelligent Edge platform that is operating system, processor architecture, and hardware agnostic.

As such, Azure IoT Edge is a fully managed service built on Azure IoT Hub that allows you to remotely manage code on your devices so that you can send more of your (cloud) workloads to the edge devices - Artificial Intelligence (AI), Azure and third-party services, or your own business logic - to run on these devices via standard

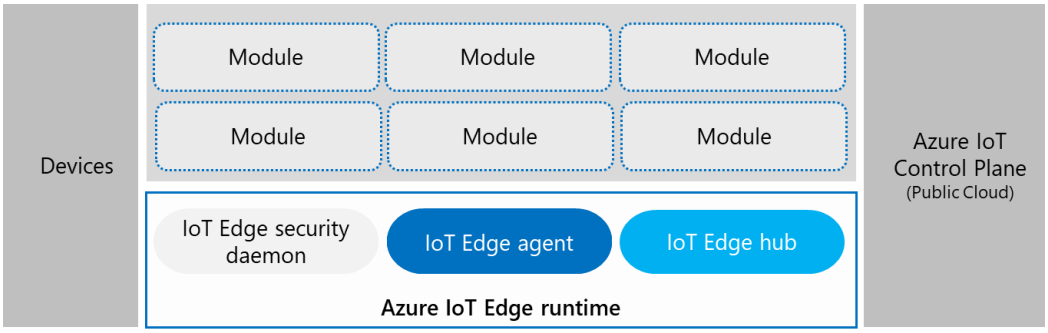
[47] Securing the Intelligent Edge: https://azure.microsoft.com/en-us/blog/securing-the-intelligent-edge/

containers. By moving certain workloads to the edge of the network, your devices spend less time communicating with the cloud, react more quickly to local changes, and operate reliably even in extended offline periods.

> **Note** For more information, see the Microsoft Azure IoT Architecture Reference[48] guide. This guide aims to accelerate customers building IoT solutions on the Azure IoT platform, and more generally speaking on Azure, by providing a proven production ready architecture, with proven technology implementation choices, and with links to Solution Accelerator reference architecture implementations such as Remote Monitoring[49] and Connected Factory[50] on GitHub.
>
> This document offers an overview of the IoT space, recommended subsystem factoring for scalable IoT solutions, prescriptive technology recommendations per subsystems, and detailed sections per subsystem that explore use cases and technology alternatives.

As such, Azure IoT Edge is an Open Source Project available on GitHub[51] that can be deployed on both Linux (ARM & x64) and Microsoft Windows operating systems. Whether it is to monitor surveillance at home or for large production lines, Azure IoT Edge can run on a resource-constrained device such as ARM32 devices as well as on a powerful x86 server running in an enterprise data center. This flexibility makes it really game-changing in the industry of edge computing.

Let's take a closer look on how an Azure IoT Edge works:



It consists mainly of those same components:

- **Devices**. Sensors or actuators that collect datas. they are not directly connected to the cloud. Each device has a corresponding module registered within the edge layer. These modules are instantiated as containers that are managed by the runtime agent.
- **Modules**. Docker containers that contain codes to be run on devices. Modules are built from standard Dockerfile definitions and pushed to a public or private registry. Thanks to the runtime agent, they can interact with each other.

---

[48] MICROSOFT AZURE IOT ARCHITECTURE REFERENCE GUIDE: http://download.microsoft.com/download/A/4/D/A4DAD253-BC21-41D3-B9D9-87D2AE6F0719/Microsoft_Azure_IoT_Reference_Architecture.pdf

[49] Remote Monitoring Solution with Azure IoT: https://github.com/Azure/azure-iot-pcs-remote-monitoring-dotnet/

[50] Azure IoT connected factory preconfigured solution: https://github.com/Azure/azure-iot-connected-factory

[51] The IoT Edge OSS project: https://github.com/Azure/iotedge

- **Azure IoT Edge runtime**. Establishes a secure connection with Azure IoT (public cloud) and manages the interactions between modules. It has three components:
    a. IoT Edge security daemon[52] starts each time an Azure IoT Edge device boots and bootstraps the device by starting the IoT Edge agent.
    b. **IoT Edge agent**. Facilitates deployment and monitoring of modules on the Azure IoT Edge device, including the IoT Edge hub. Responsible for downloading the deployment manifest from the cloud and maintaining the desired state of configuration of the edge device. It pulls all the container images from registries and runs them based on the predefined configuration.
    c. **IoT Edge hub**. Manages communications between modules on the Azure IoT Edge device, and between the device and your Azure IoT Hub. Similar to the IoT Hub in the public cloud, it can send telemetry data to the hub that will forward it to the upstream components which are other modules defined as a part of the manifest. Edge Hub acts as a communication broker facilitating local device communication. It supports standard protocols of IoT Hub including AMQP, MQTT, etc.
- **Azure IoT Control Plane**. A cloud-based interface enables you to remotely monitor and manage IoT Edge devices.

Step-by-step directions

This module covers the following activities:

1. Creating an Azure IoT Hub.
2. Registering an Azure IoT Edge device to your Azure IoT Hub.
3. Installing and starting the Azure IoT Edge runtime on your edge device.
4. Completing the configuration of your edge device.

Each activity is described in order in the next sections.

You can either manually set up an Azure IoT Hub on Azure Portal or by running the notebook *01-configure-iot-hub.ipynb*. (The notebook will instantly create an Azure IoT Hub and an Edge device so you can jump right to the next section.)

Creating an Azure IoT Hub

The steps in this section might be typically performed by cloud developers.

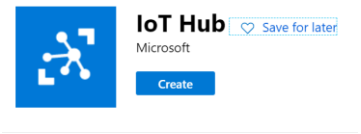
To manually set up an Azure IoT Hub, perform the following steps:

Note	For more information, see article CREATE AN IOT HUB USING THE AZURE PORTAL[53].

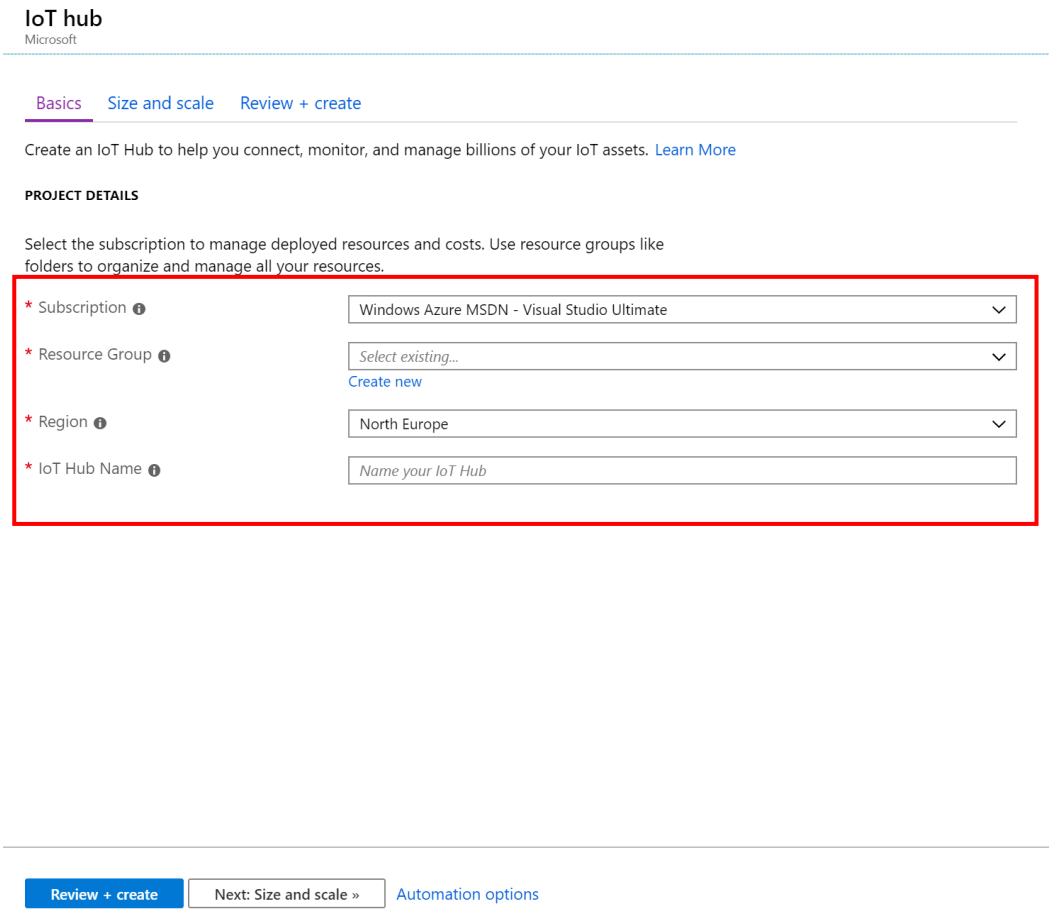
[52] AZURE IOT EDGE SECURITY MANAGER: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-security-manager

[53] CREATE AN IOT HUB USING THE AZURE PORTAL: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-create-through-portal

1. Sign in to the Azure portal at https://portal.azure.com.
2. In the left pane, select **Create a resource**. Search for "*IoT Hub*" in the **Search the Marketplace** search bar.



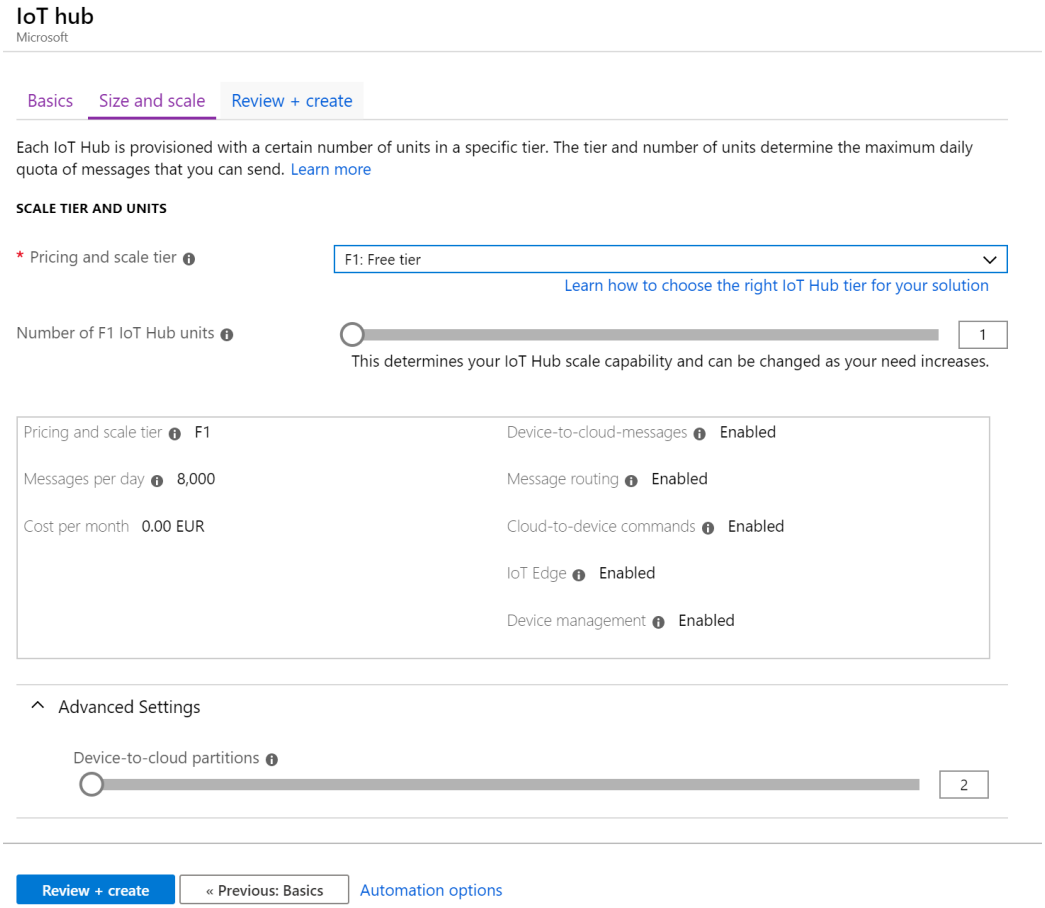
3. Select **Create.**
4. Complete the following settings.



Setting	Description
Subscription	Select the subscription to use for your Azure IoT hub.
Resource Group	You can create a new resource group or use an existing one. To create a new one, click **Create** new and fill in the name you want to use. To use instead an existing resource group, click **Use existing** and select the resource group from the dropdown list.

Region	This is the region in which you want your hub to be located. Select the location closest to you from the dropdown list.
Azure IoT Hub Name	Put in the name for your Azure IoT Hub. This name must be globally unique. If the name you enter is available, a green check mark appears.

5. Click **Next: Size and scale** to continue creating your Azure IoT Hub.



Setting	Description
Pricing and scale tier	Specify the tier to use. You can choose from several tiers depending on how many features you want and how many messages you send through your solution per day. The free level of Azure IoT Hub works for this guide. If you've used Azure IoT Hub in the past and already have a free hub created, you can use that Azure IoT hub. Keep in mind that each subscription can only have one free IoT hub. Select **F1: Free tier** or **S1: Standard tier** for the pricing tier.
Number of F1|S1 Azure IoT Hub units	Specify the number of units. The number of messages allowed per unit per day depends on your hub's pricing tier. You don't need more than one unit for this guide.
Device-to-cloud partition	Specify the number of partitions. This property relates the device-to-cloud messages to the number of simultaneous readers of the messages. Most Azure IoT Hubs only need four partitions, you can keep it by default.

6. Select **Review + Create** to review your choices.
7. Click on **Create** to create your new Azure IoT hub. Creating the hub takes a few minutes.

This is it! Your Azure IoT Hub is ready, let's now set up and connect your edge device to your Azure IoT Hub.

Registering an Azure IoT Edge device to your Azure IoT Hub

The steps in this section might be typically performed by cloud developers.

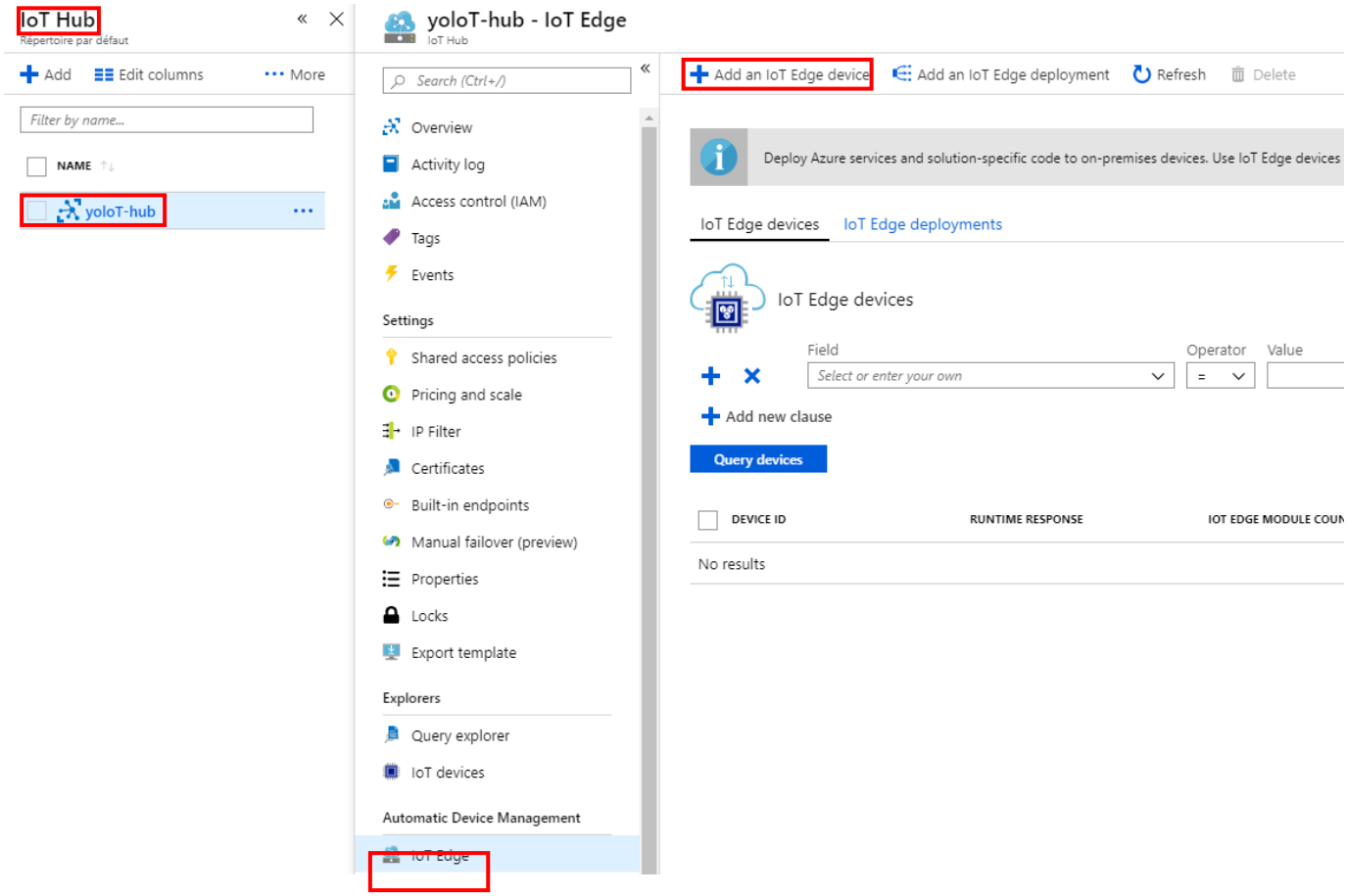
Let's now create a device identity for your Azure IoT Edge device on your Azure IoT Hub, so that it can communicate with your Azure IoT Hub. The device identity lives in the cloud, and you use a unique device connection string to associate a physical device to a device identity.

> **Note** For more information, see article REGISTER A NEW AZURE IOT EDGE DEVICE FROM THE AZURE PORTAL[54].

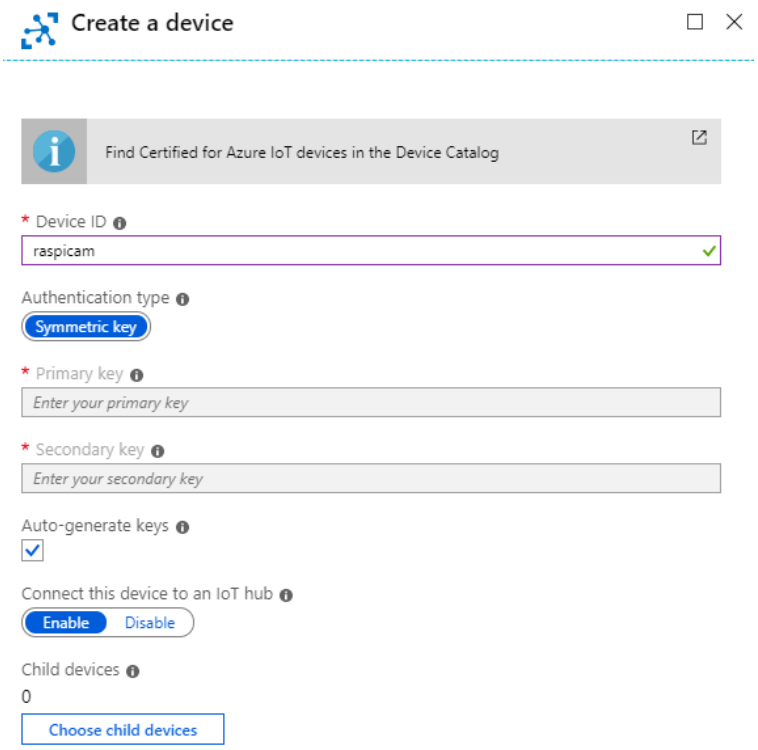
To manually create an Azure IoT Edge device, perform the following steps:

1. From Azure portal, search for "*IoT Hub*" in the search bar, select your Azure IoT Hub, and then **IoT Edge** section.

[54] REGISTER A NEW AZURE IOT EDGE DEVICE FROM THE AZURE PORTAL: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-register-device-portal

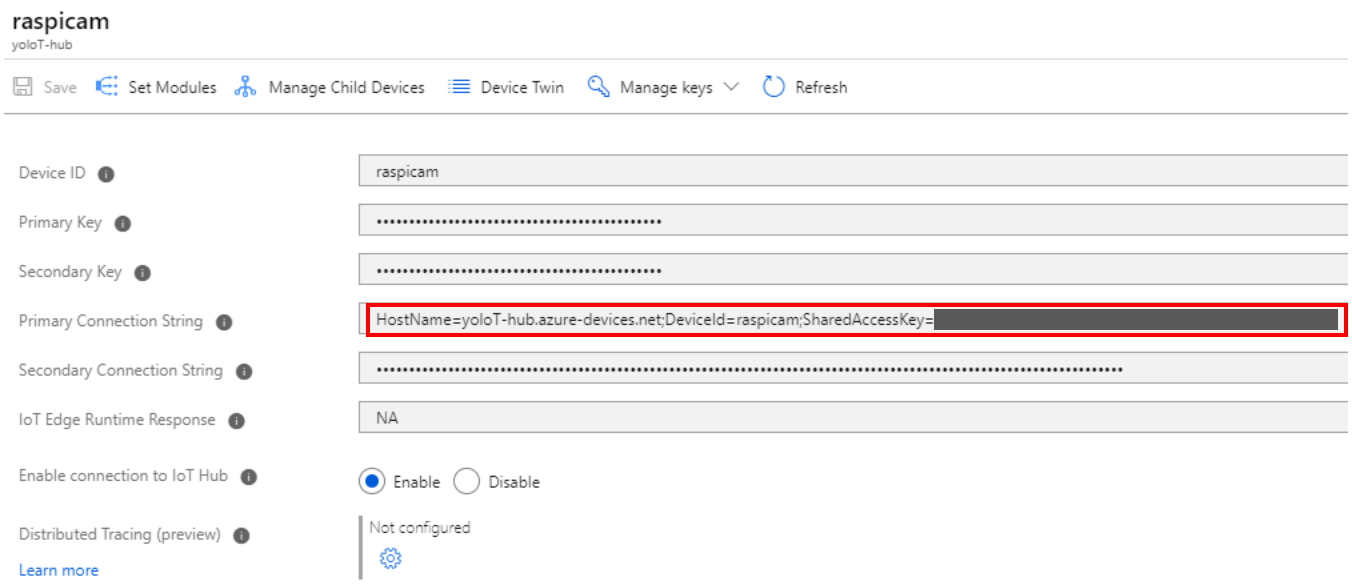


2. Select **Add an IoT Edge Device.**



3. Enter a **Device ID** and select **Save.**
4. Now click on the freshly created device. You should look out for Primary Connection String. This string is important for the rest of the guide, save it, it should look like this:

```
HostName=YourIoTHubName.azure-
devices.net;DeviceId=SimulatedDevice;SharedAccessKey={YourSharedAccessKey}
```



Now you are all set! Let's now configure the edge device.

Installing and starting the Azure IoT Edge runtime on your edge device

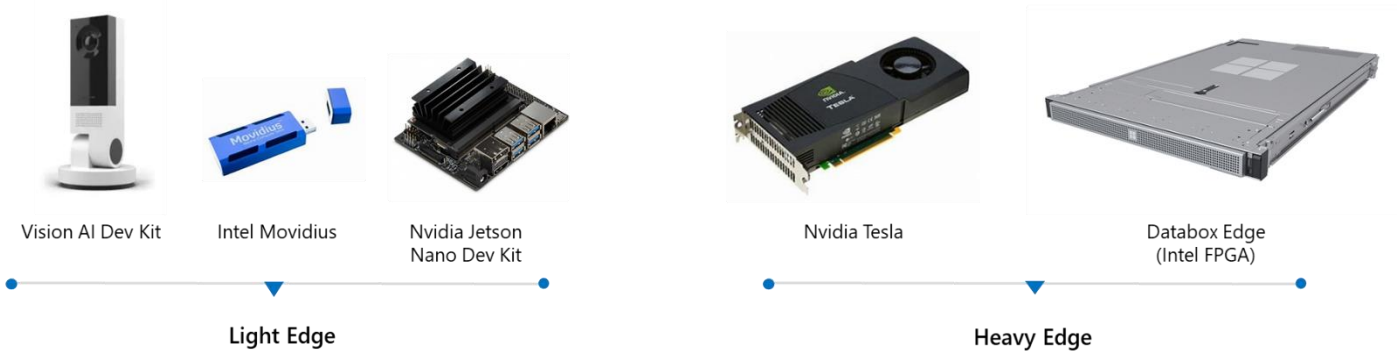
The steps in this section might be typically performed by device developers.

An intelligent edge device can typically be the Vision AI Developer Kit[55], a smart camera. This reference kit enables you to build vision AI solution and run your AI models on the device. For that purpose, the Vision AI Developer Kit works in conjunction with Azure IoT Edge and Azure Machine Learning.

A whole range a ready-to-user certified Azure IoT Edge devices are available in the Azure Certified for IoT Device Catalog[56].

[55] Vision AI Developer Kit: https://azure.github.io/Vision-AI-DevKit-Pages/

[56] Azure Certified for IoT Device Catalog: https://catalog.azureiotsolutions.com/



Using a Raspberry Pi 3

However, as mentioned earlier, for the sake of this guide, and to lessens the barrier for this walkthrough, not requiring some specific hardware, let's start by using first a Rapberry Pi 3 Model B device as your IoT Edge device.

If you just got a brand new Raspberry Pi 3, we suggest you take look at this tutorial[57] to set it up first.

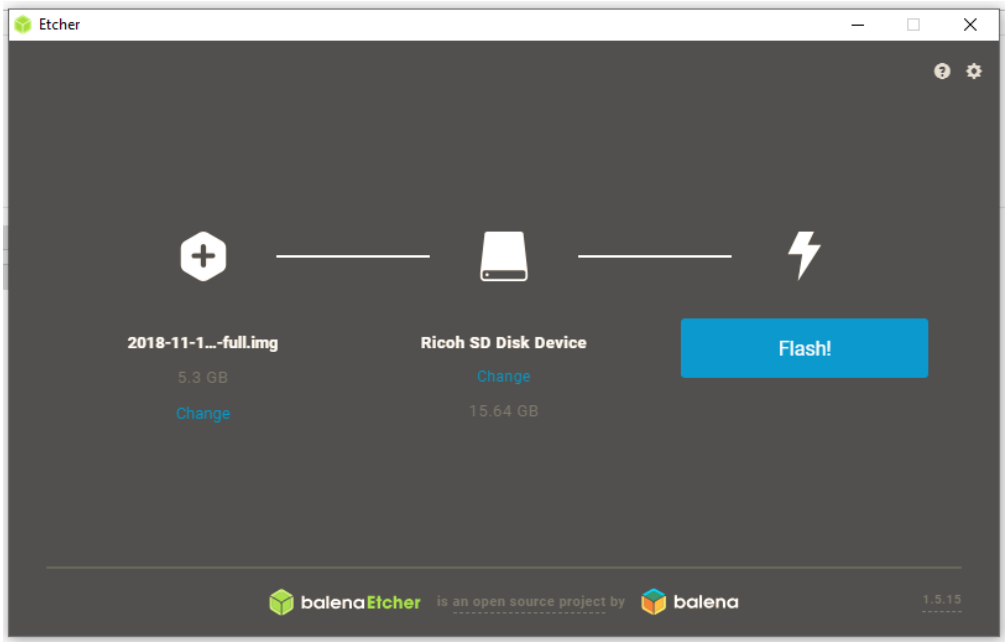
The most important steps you need to follow are:

1. Install Raspbian (for this guide, you need to install **Raspbian Stretch** which is the most stable version of Raspbian to date and the only version of Raspbian that has been fully tested with iotedge, but soon, you will need to adapt for **Buster**). You can grab the image here[58] and then you will need Etcher[59] to flash the image into a bootable Micro SD card as follows.

[57] RASPBERRY PI SETUP - A GETTING STARTED GUIDE FOR RASPBERRY PI DEVELOPMENT: https://blog.jongallant.com/2017/11/raspberrypi-setup/

[58] Debian Stretch with Raspberry Pi Desktop: https://www.raspberrypi.org/downloads/raspberry-pi-desktop/

[59] balenaEtcher: https://www.balena.io/etcher/



2. Enable SSH (to have access to your Raspberry Pi 3 remotely from your development machine). To get your raspberry pi IP address, you need to run ipconfig in your terminal and look for eth connections.
3. Update Raspbian by running in your terminal:

```
sudo apt update && sudo apt full-upgrade -y
```

When you have performed all the tasks above, you would have all the requirements to configure your Raspberry Pi 3 with Azure IoT Edge, and connect it to your Azure IoT Hub.

As such, the Azure IoT Edge runtime introduced above is deployed on all Azure IoT Edge devices. During the runtime configuration, you have to provide a device connection string.

To do that, perform the following steps:

Note For more information, see article INSTALL THE AZURE IOT EDGE RUNTIME ON DEBIAN-BASED LINUX SYSTEMS[60].

1. To register Microsoft key and software repository feed, SSH into your RPi or run in the terminal:

```
curl https://packages.microsoft.com/config/debian/stretch/multiarch/prod.list > ./microsoft-prod.list
sudo cp ./microsoft-prod.list /etc/apt/sources.list.d/
curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > microsoft.gpg
sudo cp ./microsoft.gpg /etc/apt/trusted.gpg.d/
```

[60] INSTALL THE AZURE IOT EDGE RUNTIME ON DEBIAN-BASED LINUX SYSTEMS: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-install-iot-edge-linux

Note Microsoft builds and supports a variety of software products for Linux systems and makes them available via standard APT and YUM package repositories. For that, they need to be configured by installing the Linux package that matches your linux distribution (here Raspbian Stretch). The package will install the repository configuration along with the GPG public key used by tools such as apt/yum/zypper to validate the signed packages and/or repository metadata.

2. To install the container runtime, run:

```
sudo apt-get update
sudo apt-get install moby-engine
sudo apt-get install moby-cli
```

   Moby is the only container engine that officially supports Azure IoT Edge. You might also need Moby command-line interface for development.

3. To install the Azure IoT Edge Security Daemon[61]:

   a. Run:

```
sudo apt-get update
sudo apt-get install iotedge
```

   b. Once everything is installed, you need to configure the Daemon. Make sure that docker is installed (which comes with the installation of moby-engine), check it by running:

```
sudo docker version
```

All that's left to do now is to connect the device to your Azure IoT Hub.

To do that, perform the following steps:

1. Open up */etc/iotedge/config.yaml* in your editor of choice, for example nano:

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

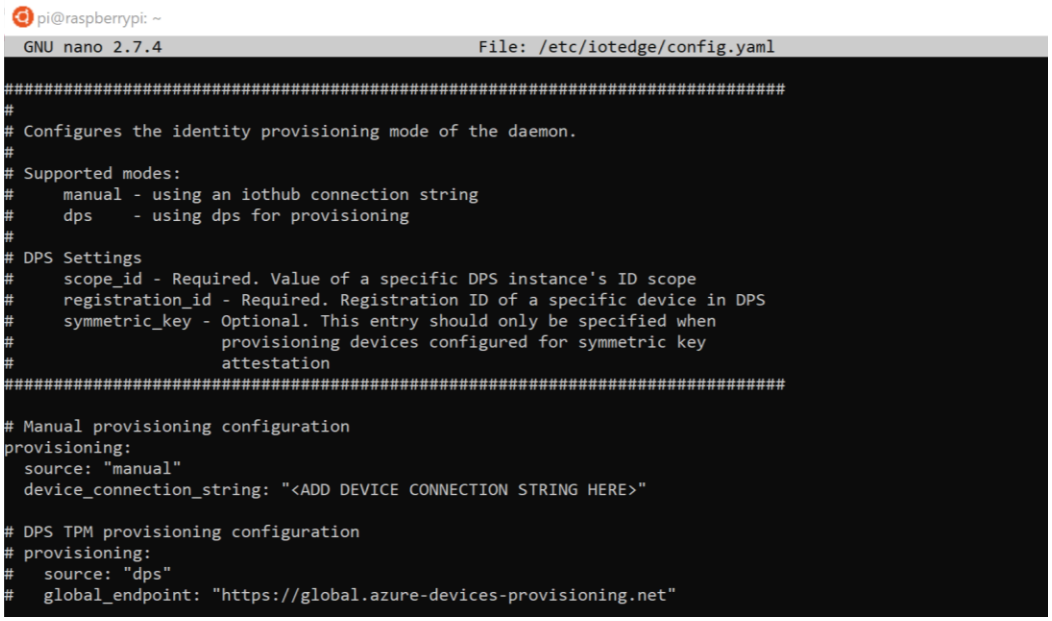
2. Azure IoT Edge device can be provisioned manually using a device connection string provided by Azure IoT Hub or automatically using the Device Provisioning Service (DPS)[62], which is helpful when you have many devices to provision. For the sake of simplicity here, you will provision your device manually (since you only deal with a single device here).

   To do that, you need to provide it with the device connection string (saved from the last section) which is of the form:

```
HostName=YourIoTHubName.azure-
devices.net;DeviceId=SimulatedDevice;SharedAccessKey={YourSharedAccessKey}
```

[61] AZURE IOT EDGE SECURITY MANAGER: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-security-manager

[62] PROVISIONING DEVICES WITH AZURE IOT HUB DEVICE PROVISIONING SERVICE: https://docs.microsoft.com/en-us/azure/iot-dps/about-iot-dps



Here just replace <ADD DEVICE CONNECTION STRING HERE> by the above device string connection.

3. Save the configuration file without changing its name (CTRL + X, Y, ENTER). Now you need to restart the runtime:

```
sudo systemctl restart iotedge
```

4. Verify the status with:

```
sudo systemctl status iotedge
```

```
pi@raspberrypi: ~
● iotedge.service - Azure IoT Edge daemon
   Loaded: loaded (/lib/systemd/system/iotedge.service; enabled; vendor preset: enabled)
   Active: active (running) since Tue 2019-08-20 14:33:47 CEST; 5min ago
     Docs: man:iotedged(8)
 Main PID: 1212 (iotedged)
    Tasks: 11 (limit: 4915)
   Memory: 12.1M
      CPU: 6.465s
   CGroup: /system.slice/iotedge.service
           └─1212 /usr/bin/iotedged -c /etc/iotedge/config.yaml

Aug 20 14:39:41 raspberrypi iotedged[1212]: 2019-08-20T12:39:41Z [INFO] - [work] - - - [2019-08-20 12:39:41.4514
Aug 20 14:39:41 raspberrypi iotedged[1212]: 2019-08-20T12:39:41Z [INFO] - [work] - - - [2019-08-20 12:39:41.5726
Aug 20 14:39:41 raspberrypi iotedged[1212]: 2019-08-20T12:39:41Z [INFO] - [work] - - - [2019-08-20 12:39:41.6886
Aug 20 14:39:42 raspberrypi iotedged[1212]: 2019-08-20T12:39:42Z [INFO] - [work] - - - [2019-08-20 12:39:42.2016
Aug 20 14:39:45 raspberrypi iotedged[1212]: 2019-08-20T12:39:45Z [INFO] - [work] - - - [2019-08-20 12:39:45.0141
Aug 20 14:39:45 raspberrypi iotedged[1212]: 2019-08-20T12:39:45Z [INFO] - [work] - - - [2019-08-20 12:39:45.2575
Aug 20 14:39:45 raspberrypi iotedged[1212]: 2019-08-20T12:39:45Z [INFO] - [work] - - - [2019-08-20 12:39:45.4594
Aug 20 14:39:45 raspberrypi iotedged[1212]: 2019-08-20T12:39:45Z [INFO] - [work] - - - [2019-08-20 12:39:45.6210
Aug 20 14:39:45 raspberrypi iotedged[1212]: 2019-08-20T12:39:45Z [INFO] - [work] - - - [2019-08-20 12:39:45.7632
Aug 20 14:39:45 raspberrypi iotedged[1212]: 2019-08-20T12:39:45Z [INFO] - [work] - - - [2019-08-20 12:39:45.8548
~
~
~
~
~
~
~
~
lines 1-21/21 (END)
```

Completing the configuration of your edge device

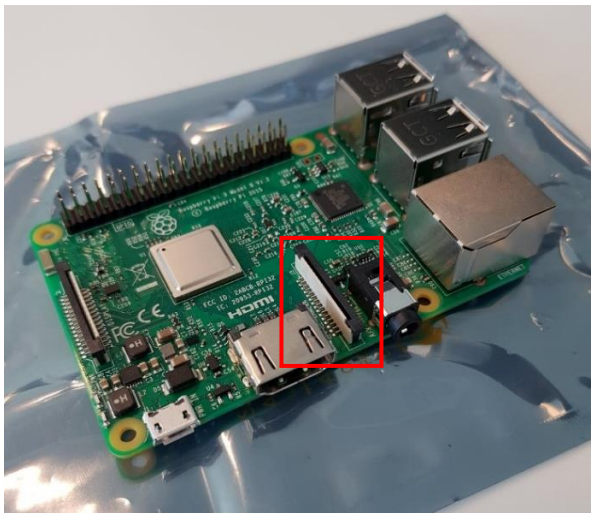
The steps in this section might be typically performed by device developers.

Raspberry Pi 3 related considerations

To complete the configuration of your Raspeberry Pi edge device, you will simply connect the Pi Camera to it to collect captures.

To set up physically your Pi Camera, perform the following steps:

1. On your Raspberry Pi, locate the slot dedicated to the camera as below.



2. Now, pull gently the top of the slot to enable insertion.

3. Insert the cable into the Raspberry Pi. The cable slots into the connector situated between the Ethernet and HDMI ports, with the silver connectors facing the HDMI port.

4. Push the top of the slot to finally lock the camera.

Now, to enable connection with the camera, you need to set up few things so let's boot up the Raspberry Pi.

If you are using Raspbian Stretch Lite, perform the following steps:

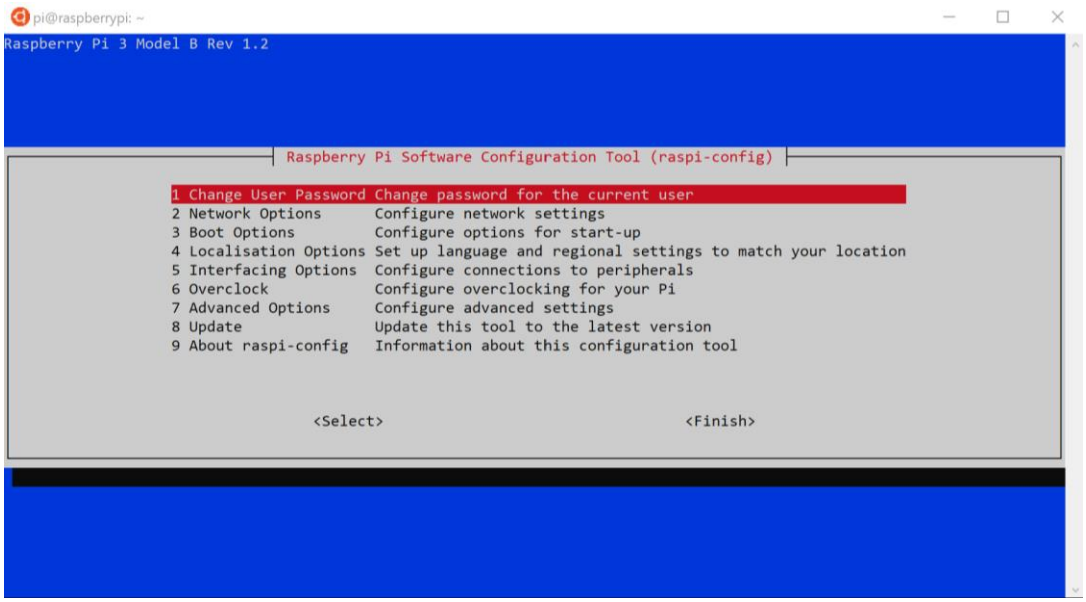
1. ssh into the device and run the following commands (otherwise, just open up a terminal):

```
sudo raspi-config
```

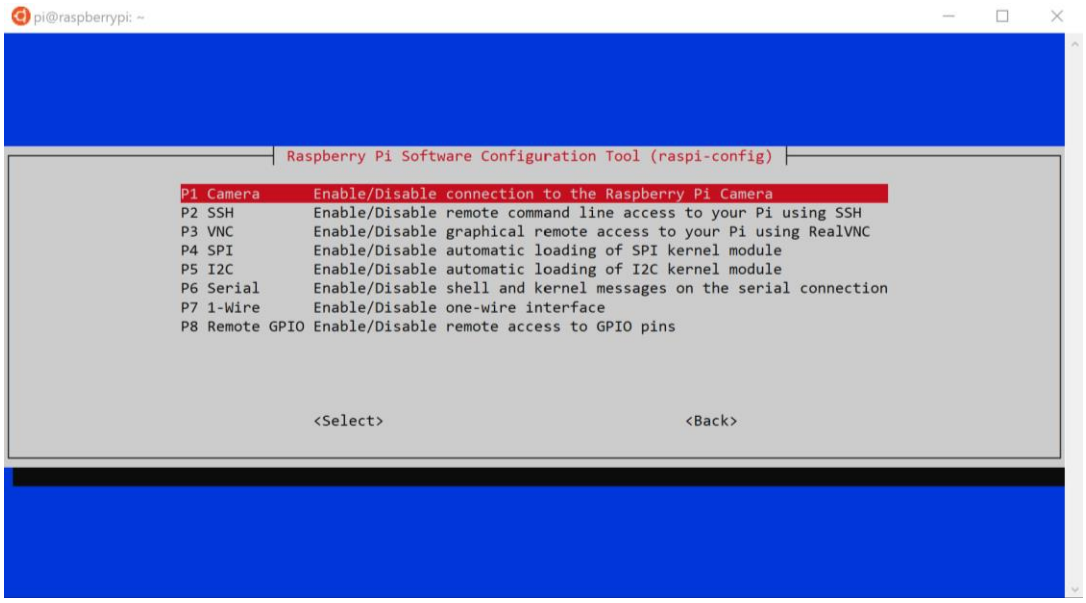
2. If the "camera" option is not listed, run:

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

3. Once everything is upgraded, run *sudo raspi-config* again:



4. Now navigate to the "Camera" option and enable it (make sure that SSH is enabled too).

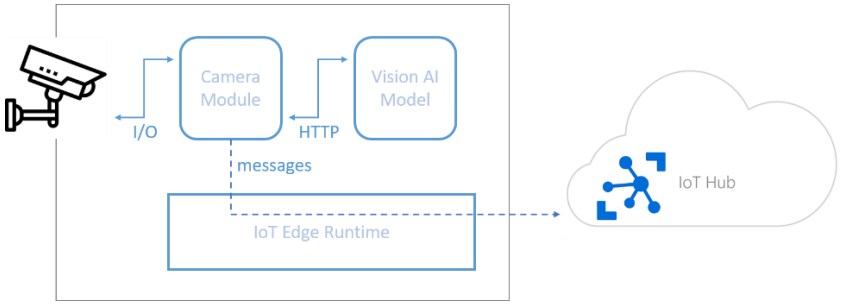


You're all set! Let's deals with the modules for your edge device.

Module 2: Building your Azure IoT Edge modules

Overview

For the sake of simplicity, and as already presented, you will be building in this guide an IoT solution that contains a camera module and another one that contains a computer vision model that perfoms car detection.



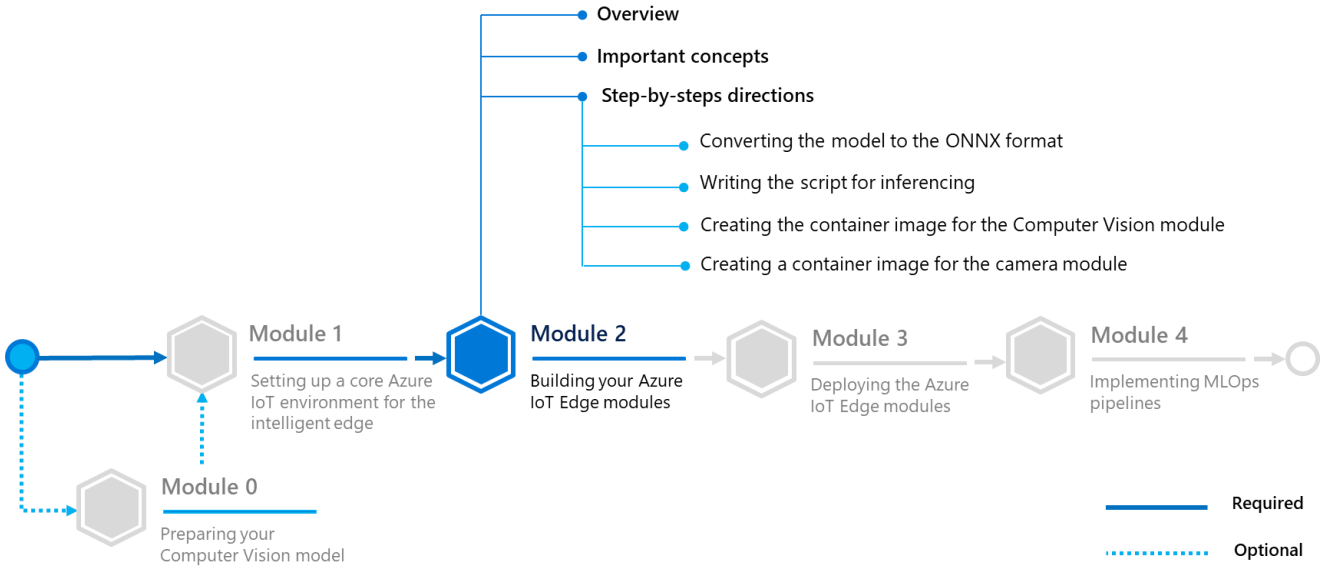
In order to process datas that are collected through the camera module, you will set up a local web service directly on the edge that contains a computer vision model capable of performing the tasks you ask for. Just like a REST API hosted directly on the edge device.

This has the advantage to be much faster for inferencing and by moving workloads to the edge, this will result in a reduced bandwidth consumption and less interaction with the cloud.

The structure of this IoT solution is such that it is very easy to adapt the computer vision model for any purpose you have in mind. As you will go through the different steps below, you will realize how easy it is to customize your model and to adapt to your needs.

In this module, you will learn to:

- Plug your own custom vision AI model without changing the architecture of the solution.
- Create a custom vision module ready to be deployed.
- Cerate a camera module ready to deployed.



Before builing the above modules, let's first consider the Open Neural Network eXchange[63] (ONNX) project that constitutes is a huge improvement in the field of AI.

Important concepts

Open Neural Network eXchange (ONNX) format

ONNX

"The Open Neural Network eXchange (ONNX) "is an open format to represent deep learning models. With ONNX, developers can move models between state-of-the-art tools and choose the combination that is best for them. ONNX is developed and supported by a community of partners"[64]

At the time of this writing, ONNX is joining[65] the LF AI Foundation[66], an umbrella foundation of the Linux Foundation supporting open source innovation in Artificial Intelligence (AI), Machine Learning, and Deep Learning.

For a long time, there was not a universal way to export Deep Learning models from one framework to another, data scientists who worked with Tensorflow/Keras couldn't exchange their models with those who worked with Pytorch. Most of the time, you needed to rewrite almost all the code from one framework to another, which means the whole structure of the neural network. Furthermore, models could not be easily used to inference by

[63] Open Neural Network eXchange (ONNX): https://onnx.ai/

[64] Ibid

[65] ONNX JOINS LINUX FOUNDATION: https://cloudblogs.microsoft.com/opensource/2019/11/14/onnx-joins-linux-foundation/

[66] LF AI Foundation: https://lfai.foundation/?p=965

another framework, making cooperation between data scientists difficult if they were not using the same framework.

In 2017, from a close cooperation between Microsoft and Facebook, ONNX was born to address the issues above. It is designed to be a universal format to store what's essential in a neural network since each framework has its own particularities. Pytorch uses, for instance, dynamic graphs while Tensorflow uses static graphs.

Here are the advantages of ONNX:

- **Framework interoperability**. It provides a free hand to the data scientists to train a model in a framework and to generate inference in another framework.
- **Hardware optimizations**. It is easy to give the ability of optimization to the data scientists. It provides the benefits of ONNX compatible runtimes and libraries to every tool which uses the exported ONNX models, and it results in the maximization of the performance.

Depending on which deep learning framework you are working on, there are converters from one to another.

As of this writing, the supported frameworks are as follows:

Note For more information, you can check out here[67] sample scripts to convert models from one framework to another.

Open Neural Network eXchange (ONNX) runtime

Let's clarify the second point. In fact, ONNX comes with the ONNX Runtime[68], an optimized AI inference engine for ONNX models. Launched in 2018, Intel and Microsoft have been working hand in hand to deliver powerful development tools to take advantage of Intel's latest AI-accelerating technologies across the intelligent cloud and the intelligent edge.

[67] ONNX Converter Scripts: https://github.com/onnx/onnx-docker/tree/master/onnx-ecosystem/converter_scripts

[68] ONNX Runtime: cross-platform, high performance scoring engine for ML models: https://github.com/microsoft/onnxruntime

Optimized for both cloud and edge inferencing, the ONNX Runtime expands the range of Computer Vision models that can be deployed at the edge (especially resource-constrained devices that can be run within a reasonable speed).

Note For more information, see session BRK3012[69] "Open Neural Network Exchange (ONNX) in the enterprise: how Microsoft scales ML across the world and across devices" at the Microsoft Build 2019 Conference.

Step-by-step directions

This module covers the following activities:

1. Converting the model to ONNX format.
2. Writing the script for inferencing.
3. Creating the container image for the Computer Vision module.
4. Creating a container image for the camera module.

Each activity is described in order in the next sections.

Converting the model to ONNX format

The steps in this section might be typically performed by data scientists.

At this stage, you already have a YOLOv3 model trained and saved from keras.

In this section, you will convert it to the ONNX format. To convert the YOLOv3 model from keras, there is a little more work to do (the keras2onnx package is not enough).

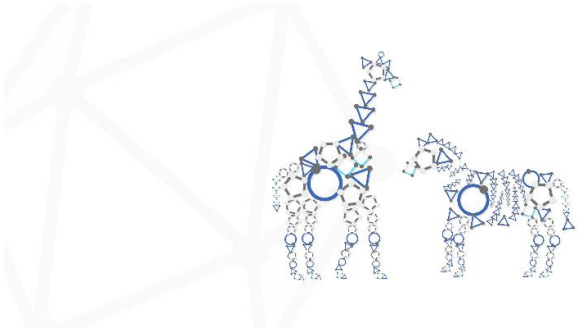
Thankfully, ONNX (from the keras-onnx repository here[70]) has released a converter for the keras YOLOv3 model you used in the module 0 (see section § **Module 0: Preparing your Computer Vision model (optional)**), you just have to run the python script *convert_yolov3_to_onnx.py* to convert your trained model to ONNX format. (Look at the *README.md* for more details.)

Also, the community supporting ONNX is keeping up-to-date a collection of state-of-the art Deep Learning pre-trained models successfully converted to ONNX format. Hence, you can find on ONNX Model Zoo[71], a collection of pre-trained, state-of-the-art models in the ONNX format, multiple well-known deep learning models such as Resnet50, GoogleNet, SSD, etc. ready-to-use.

[69] BRK3012 "Open Neural Network Exchange (ONNX) in the enterprise: how Microsoft scales ML across the world and across devices": https://mybuild.techcommunity.microsoft.com/sessions/76978?source=sessions#top-anchor

[70] keras-onnx yolo3: https://github.com/onnx/keras-onnx/tree/master/applications/yolov3

[71] ONNX Model Zoo https://github.com/onnx/models



ONNX Model Zoo, a collection of pre-trained models for state-of-the-art models in deep learning

If you don't want to train your own model, you can pick up one of the pre-trained model and use it for the rest of this guide since they are all successfully converted models that are ready to be used.

For the sake of simplicity, you will be using the YOLOv3 model available on ONNX Model Zoo. The particularity of this pre-trained model is that it has been trained on the COCO dataset which contains over 80 classes. For your example (the parking lot), the model can detect cars which is what you wanted.

Writing the script for inferencing

The steps in this section might be typically performed by data scientists.

For Machine Learning inference, each model has its own treatment when it comes to processing data. Writing the inferencing script consists in defining the functions to be called before data are fed into the model and after. You need to pay attention on the input and output shapes of the image arrays. Each of these points are reminded for each model available on the Model Zoo.

For instance, in the section for tiny YOLOv2:

Input

shape `(1x3x416x416)`

Preprocessing

Output

shape `(1x125x13x13)`

Postprocessing

The output is a `(125x13x13)` tensor where 13x13 is the number of grid cells that the image gets divided into. Each grid cell corresponds to 125 channels, made up of the 5 bounding boxes predicted by the grid cell and the 25 data elements that describe each bounding box (`5x25=125`). For more information on how to derive the final bounding boxes and their corresponding confidence scores, refer to this post.

Thankfully, for YOLOv3, the authors already give us the pre/postprocess functions so implementing the inferencing script is an easy task (check out *predict_yolov3.py*) but usually, the structure of the script remains the same from one model to another:

```
import … (other imports)
import onnxruntime

def init():
    global session
    session = onnxruntime.InferenceSession('your_model.onnx')

def preprocess():
    ## some preprocessing ##

def postprocess():
    ## some postprocessing ##

def predict(data):
    pre_data = preprocess(data)
    prediction = session.run(…)
    result = postprocess(prediction)
    return result
```

Note Here, you are using the ONNX Runtime to speed up ML inference, depending on which platform you are working on, importing onnxruntime may not be an easy task since the package is not built for all platforms. Especially here, you are working on ARM devices and there is no python wheel available for Raspbian.

Check out here[72] for more info on how to build manually the wheel. You may be tempted to use the python wheel in the current repository as well (it may not be the latest version, but for the sake of the guide, this one is perfectly enough).

Here is an example (from *predict_yolov3.py*):

```
from PIL import Image
import numpy as np
import sys
import os
import numpy as np
import json
import onnxruntime

# Special json encoder for numpy types
class NumpyEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, (np.int_, np.intc, np.intp, np.int8,
            np.int16, np.int32, np.int64, np.uint8,
            np.uint16, np.uint32, np.uint64)):
            return int(obj)
        elif isinstance(obj, (np.float_, np.float16, np.float32,
            np.float64)):
            return float(obj)
        elif isinstance(obj,(np.ndarray,)):
            return obj.tolist()
        return json.JSONEncoder.default(self, obj)
```

[72] BUILDING ONNX RUNTIME - GETTING STARTED: https://github.com/microsoft/onnxruntime/blob/master/BUILD.md

```
# Needed for preprocessing
def letterbox_image(image, size):
    iw, ih = image.size
    w, h = size
    scale = min(w/iw, h/ih)
    nw = int(iw*scale)
    nh = int(ih*scale)

    image = image.resize((nw,nh), Image.BICUBIC)
    new_image = Image.new('RGB', size, (128,128,128))
    new_image.paste(image, ((w-nw)//2, (h-nh)//2))
    return new_image

class YOLOv3Predict:
    def __init__(self, model, label_file):
        self.model = model
        self.label_file = label_file
        self.label = []

    def get_labels(self):
        with open(self.label_file) as f:
            for line in f:
                self.label.append(line.rstrip())

    def initialize(self):
        global session
        print('Loading model...')
        self.get_labels()
        session = onnxruntime.InferenceSession(self.model)
        print('Model loaded!')

    def preprocess(self,img):
        model_image_size = (416, 416)
        boxed_image = letterbox_image(img, tuple(reversed(model_image_size)))
        image_data = np.array(boxed_image, dtype='float32')
        image_data /= 255.
        image_data = np.transpose(image_data, [2, 0, 1])
        image_data = np.expand_dims(image_data, 0)
        return image_data

    def postprocess(self, boxes, scores, indices):
        out_boxes, out_scores, out_classes = [], [], []
        for idx_ in indices:
            out_classes.append(idx_[1])
            out_scores.append(scores[tuple(idx_)])
            idx_1 = (idx_[0], idx_[2])
            out_boxes.append(boxes[idx_1])
        return out_boxes, out_scores, out_classes

    def predict(self,image):
        image_data = self.preprocess(image)
        image_size = np.array([image.size[1], image.size[0]], dtype=np.float32).reshape(1, 2)
        input_names = session.get_inputs()
        feed_dict = {input_names[0].name: image_data, input_names[1].name: image_size}
        boxes, scores, indices = session.run([], input_feed=feed_dict)
        predicted_boxes, predicted_scores, predicted_classes = self.postprocess(boxes, scores, indice
s)
        results = []
        for i,c in enumerate(predicted_classes):
            data = {}
```

```
            data[self.label[c]] = json.dumps(predicted_boxes[i].tolist()+[predicted_scores[i]], cls=N
umpyEncoder)
            results.append(data)
        return results
```

Creating the container image for the Computer Vision module

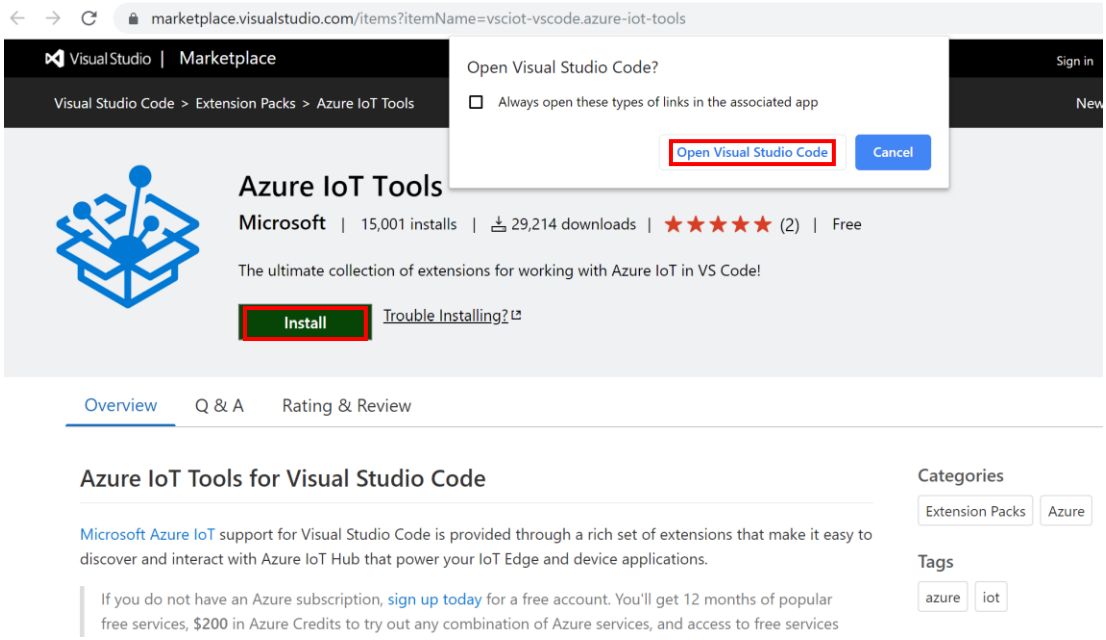
The steps in this section might be typically performed by device developers and cloud developers.

In this section, you will create the docker image that contains your REST API and you will push on Azure Container Registry to keep track of the different versions of your model and to facilitate the deployment at the edge.

What follows can be replicated for any other modules, in fact, you will see that the process is similar from one module to another. Through the development of this module, you will see how to develop a custom AI module for your IoT solution. Module development is available in Python, C, C#, Java and Node.js.

In this guide, you will be covering the development of a custom Python module for AI inferencing.

For what follows, you will need to install Visual Studio Code[73] along with the Azure IoT Tools extension pack[74]:



This extension pack will install in Visual Studio code all the functionalities you need to start developing custom IoT modules. It indeed comprises extensions make it easy to discover and interact with Azure IoT Hub that power your Azure IoT Edge device(s), provide project templates, automate the creation of the deployment manifest, and allow you to monitor and manage your Azure IoT Edge device(s):

[73] Visual Studio Code: https://code.visualstudio.com/

[74] Azure IoT Tools for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-tools

- The Azure IoT Hub Toolkit[75] extension allows you to interact with an Azure IoT Hub, manage connected Azure IoT Edge devices, and enable distributed tracing for your Azure IoT applications.
- The Azure IoT Edge for Visual Studio Code[76] allows you to easily code, build, and debug your custom logic and deploy it to your Azure IoT Edge devices.

Perform the following steps:

1. Launch Visual Studio Code.
2. select **View** > **Extensions**.
3. Search for **Azure IoT Tools**, which is actually a collection of extensions that help you interact with your IoT Hub and your devices, as well as developing IoT Edge modules.
4. Select **Install**. Each included extension installs individually.

You will need to setup up your Azure account to manage IoT Hub resources from within Visual Studio Code.

Note For more information, see article USE VISUAL STUDIO CODE TO DEVELOP AND DEBUG MODULES FOR AZURE IOT EDGE[77].

As you are developing in Python, make sure to also install the Python extension[78] (if Visual Studio Code have not already recommended it.)

Additionally, you will need Docker CE[79] configured to run Linux containers as you will be developing for **ARM32v7** devices.

Let's create a new project!

To do that, perform the following steps:

1. Open **Visual Studio Code** and go to **View > Command Palette** or press CTRL + SHIFT + P

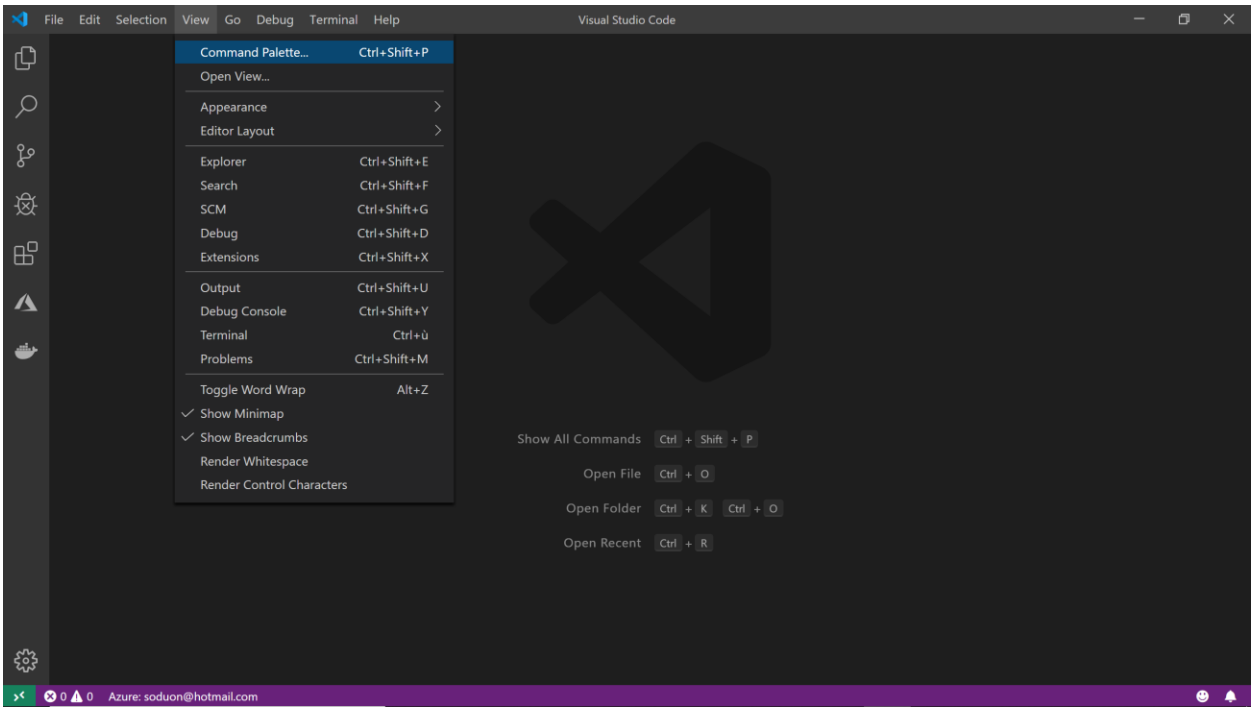
[75] Azure IoT Hub Toolkit: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-toolkit

[76] Azure IoT Edge for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=vsciot-vscode.azure-iot-edge

[77] USE VISUAL STUDIO CODE TO DEVELOP AND DEBUG MODULES FOR AZURE IOT EDGE: https://docs.microsoft.com/en-us/azure/iot-edge/how-to-vs-code-develop-module

[78] Python extension for Visual Studio Code: https://marketplace.visualstudio.com/items?itemName=ms-python.python

[79] ABOUT DOCKER ENGINE - COMMUNITY: https://docs.docker.com/install/



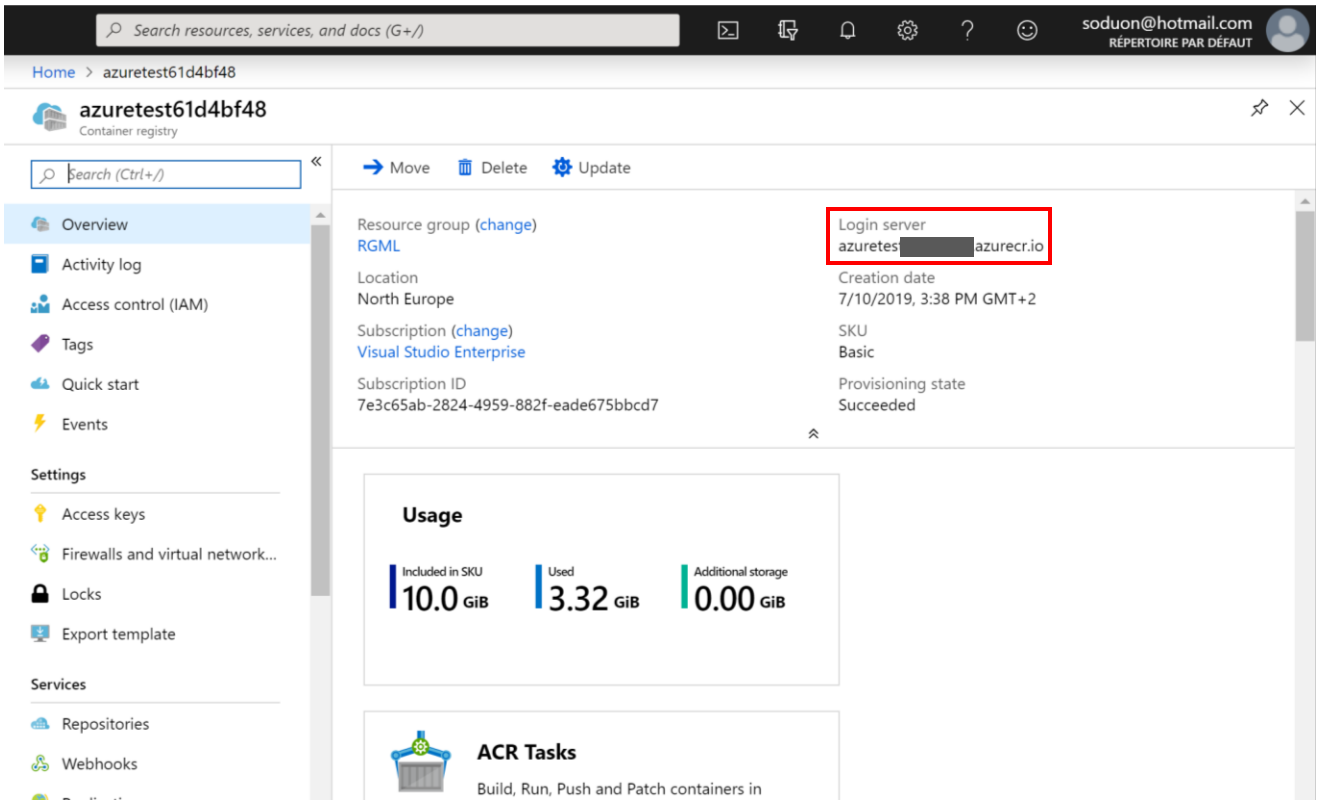
2. Search for **Azure: Sign In** to sign in to your Azure account.
3. Once signed in, create a new solution by searching **Azure IoT Edge: New IoT Edge Solution**

Settings	Options
Select folder	Choose the location on your development machine for Visual Studio Code.
Provide a solution name	Name of your solution (for example **Rpi3-objectdetection**)
Select module template	Choose **Python Module**.
Provide a module name	Name your module **PythonModule**. You need at least one custom module, for the purpose of this guide, name it **ObjectDetection**
Provide Docker image repository for the module	An image repository includes the name of your container registry and the name of your container image. Your container image is prepopulated from the name you provided in the last step. Replace **localhost:5000** with the login server value from your Azure container registry.

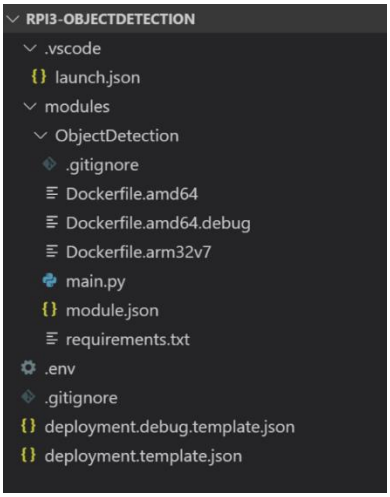
Important note You need a Docker-compatible registry to hold your container images. Two popular Docker registry services are Azure Container Registry[80] and Docker Hub[81]. When you created your Azure ML workspace (in Module 0, see section § **Module 0: Preparing your Computer Vision model (optional)**), it automatically created an associated private Azure Container registry. Azure Container Registry allows you to manage a Docker private registry in Azure where you can store and manage your private Docker container images. You can find the login server on the Overview page of your container registry in the Azure portal. The final image repository looks like <registry name>.azurecr.io/pythonmodule.

[80] Azure Container Registry: https://azure.microsoft.com/en-us/services/container-registry/

[81] Docker Hub: https://www.docker.com/products/docker-hub

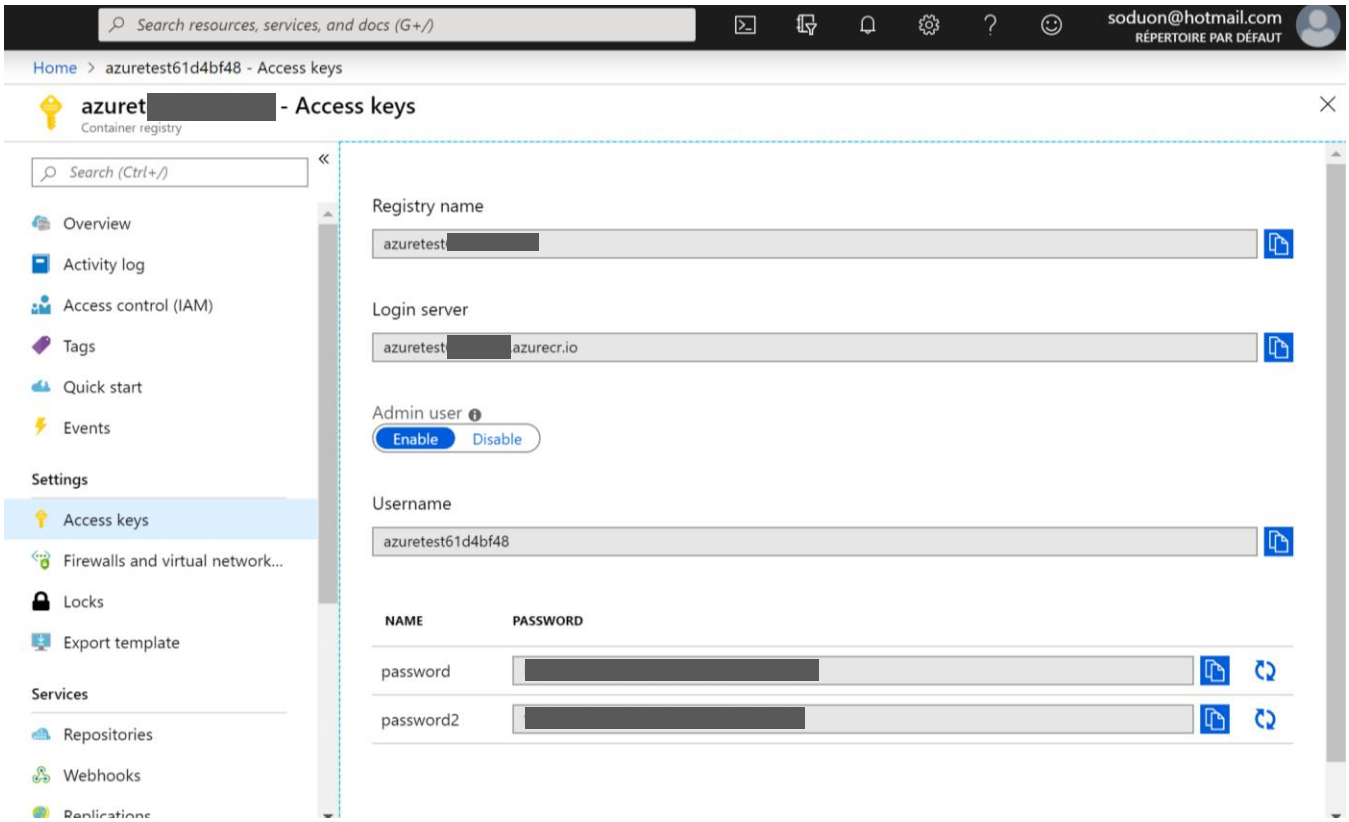


Your directory now should look like this:

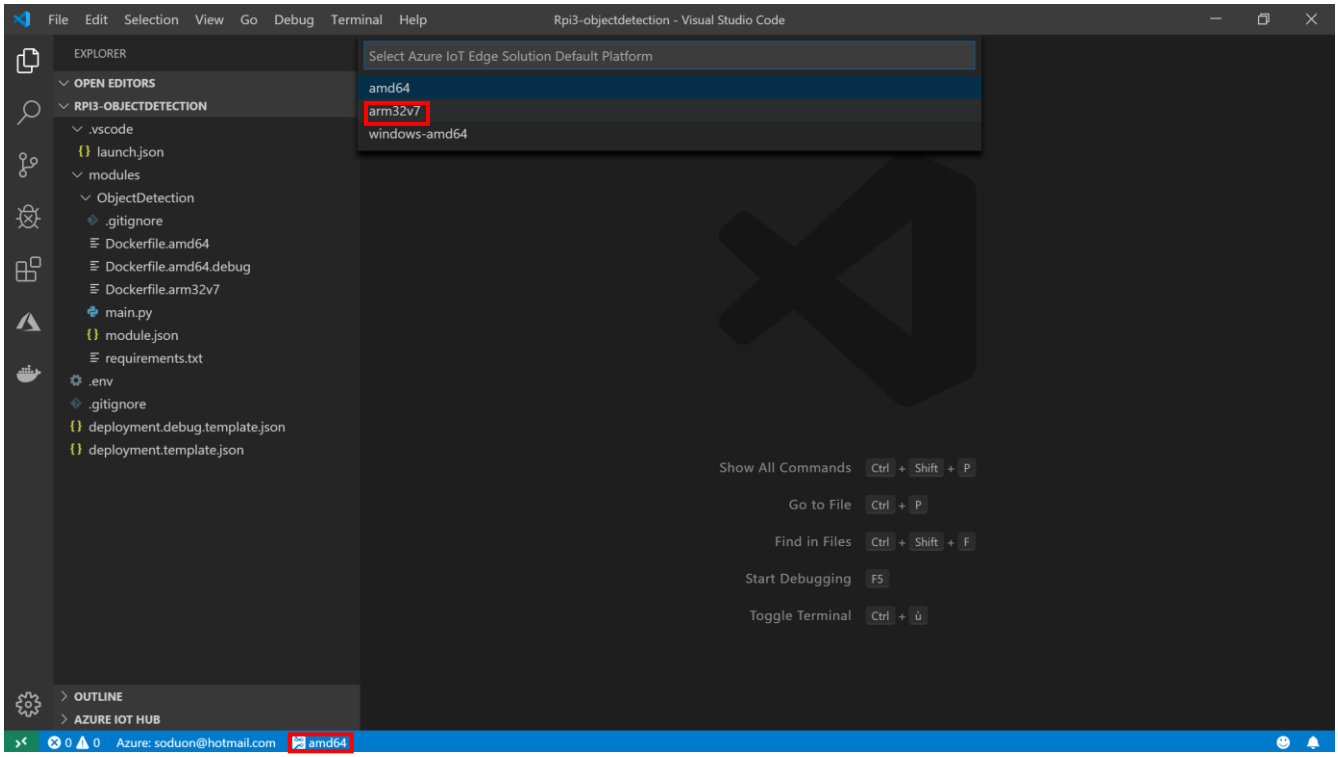


- *.Visual Studio Code* folder contains settings about your development environment (for Visual Studio Code only).
- *modules* folder contains the different modules you are going to create.
- *deployment.template.json* is a deployment manifest template that will be updated as you create more modules… it provides a manifest template that can be used to deploy custom desired properties as the development goes on.

- *.env* file contains your registry credentials, by default, Visual Studio Code automatically retrieve your credentials from your Azure account, if not, check that your username and password match with those from the Azure portal:



4. Now you need to choose a target architecture – As of this writing, Visual Studio Code supports the development on Linux ARM32v7 and AMD64 devices -. Select the button on the bottom (by default it's on **amd64**) and choose **arm32v7**.



For the sake of clarity, let's create two folders inside of the *ObjectDetection* folder: *app/* and *build/* and move the *requirements.txt* file to the *build* folder (also, to separate the arm32v7 devices from the amd64 ones, rename it *arm32v7-requirements.txt*) and the *main.py* file to the *app* folder.

5. Set up the REST API for your AI module. Copy the *predict_yolov3.py* and *labels.txt* files and the *model.onnx* AI model into the *app/* folder.

6. Create inside of the same folder a python file *app.py* and copy the following code inside:

```
import json
import os
import io

# Imports for the REST API
from flask import Flask, request # For developement
from waitress import serve # For production

# Imports for image processing
from PIL import Image

# Imports from predict_yolov3.py
from predict_yolov3 import initialize, predict_image

app = Flask(__name__)

@app.route('/')
def index():
    return 'Computer Vision module listening'

# Prediction service /image route handles either
#     - octet-stream image file
#     - a multipart/form-data with files in the imageData parameter
```

```
@app.route('/image', methods=['POST'])
def predict_image_handler():
    try:
        imageData = None
        if ('imageData' in request.files):
            imageData = request.files['imageData']
        else:
            imageData = io.BytesIO(request.get_data())
        img = Image.open(imageData)
        results = predict_image(img)
        return json.dumps(results)
    except Exception as e:
        print('EXCEPTION:', str(e))
        return 'Error processing image', 500

if __name__ == '__main__':
    # Load and intialize the model
    initialize()

    # Run the server
    print("Running the Computer Vision module...")
    #app.run(host='0.0.0.0', port=8885, debug=True) # For development
    serve(app, host='0.0.0.0', port=8885) # For production
```

What it does is that it will set up a flask REST server that will listen on port **8885** to POST requests sent from the cameraModule (see next section below).

As already introduced, IoT modules are in the form of Docker images so all that's left to do now is to write your dockerfile. As you are developing for a Raspberry Pi 3, you will be using the raspbian stretch for Rpi3 parent image from balenalib: *balenalib/raspberrypi3:stretch*

7. Open up the *Dockerfile.arm32v7* file and copy/paste the following code:

```
FROM balenalib/raspberrypi3:stretch
# The balena base image for building apps on Raspberry Pi 3.

RUN echo "BUILD MODULE: ObjectDetection"

RUN [ "cross-build-start" ]

# Install basic dependencies
RUN install_packages \
    python3 \
    python3-pip \
    python3-dev \
    build-essential \
    libopenjp2-7-dev \
    libtiff5-dev \
    zlib1g-dev \
    libjpeg-dev \
    libatlas-base-dev \
    wget

# Install Python packages
COPY /build/arm32v7-requirements.txt ./
RUN pip3 install --upgrade pip
RUN pip3 install --upgrade setuptools
RUN pip3 install --index-url=https://www.piwheels.org/simple -r arm32v7-requirements.txt
```

```
# Build ONNX Runtime from wheel (need to be updated for future versions of onnxruntime)
COPY /build/onnxruntime-0.5.0-cp35-cp35m-linux_armv7l.whl ./
RUN  pip3 install onnxruntime-0.5.0-cp35-cp35m-linux_armv7l.whl

# Cleanup
RUN rm -rf /var/lib/apt/lists/* \
    && apt-get -y autoremove

RUN [ "cross-build-end" ]

# Add the application
ADD app /app

# Expose the port for prediction
EXPOSE 8885

# Set the working directory
WORKDIR /app

# Run the flask server for the endpoints
CMD ["python3","app.py"]
```

Basically, this builds a container that will run the flask server with all the dependencies installed. Note that you will be installing the ONNX Runtime from a cross-compiled wheel (that you can find here[82]). In the future, some features might be obsolete but the current version is up and running.

8. Update the *arm32v7-requirements.txt* file and copy the python wheel of onnxruntime into the *build/* folder.

```
pillow
numpy
flask
waitress
```

Creating a container image for the camera module

The steps in this section might be typically performed by device developers.

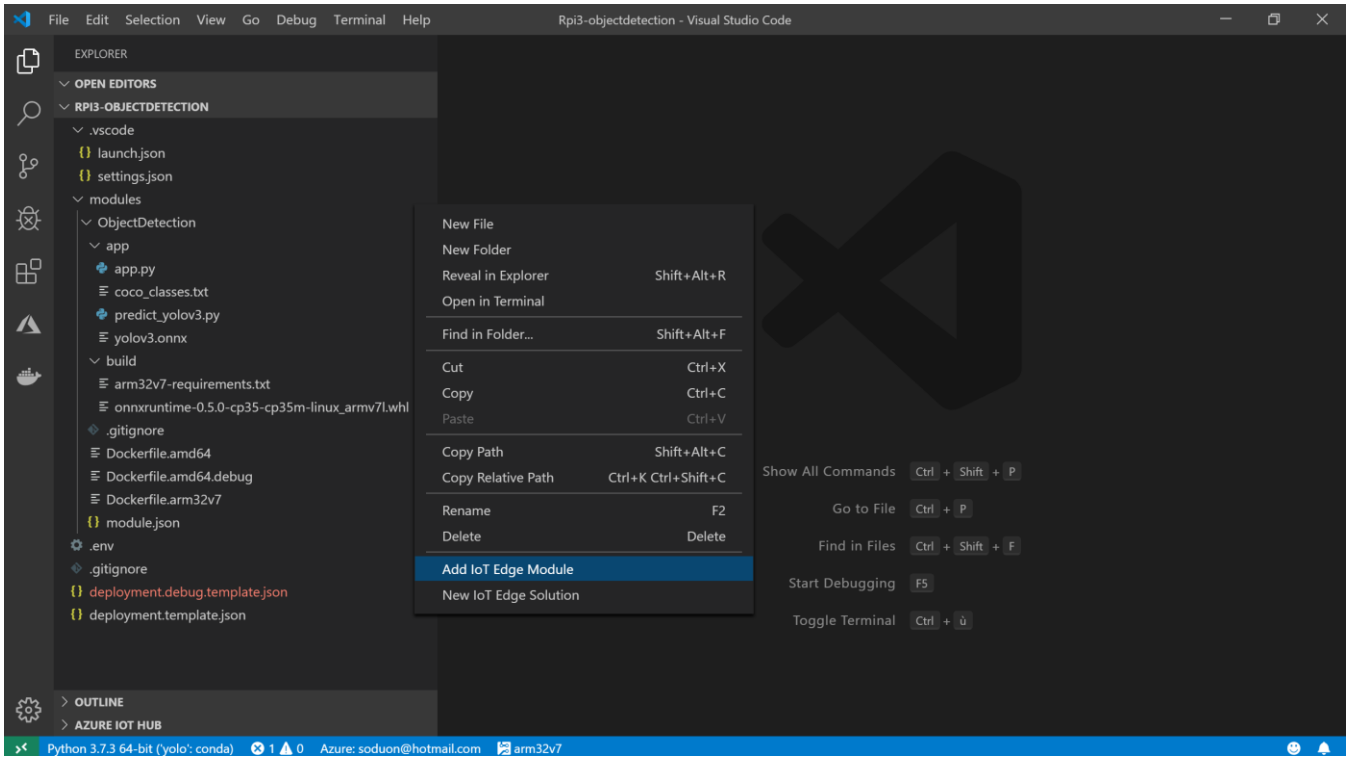
Using a Raspberry Pi 3

In order to create the camera module, you will be following the same path as for the Computer Vision module above.

[82] Onnx runtime wheel: https://github.com/lafius/azure-iot-edge-cv-model-on-raspberry-pi-samples/tree/master/IoT/Rpi3-objectdetection/modules/ObjectDetection/build

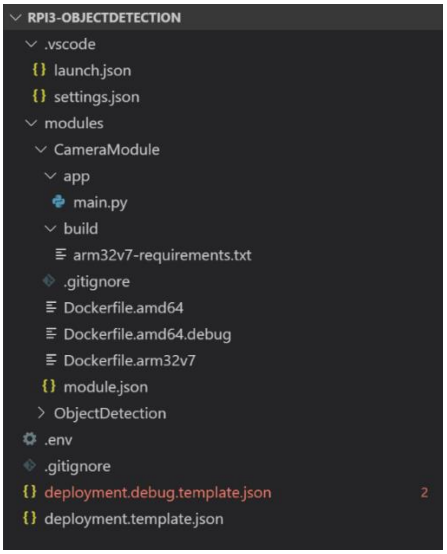
Perform the following steps:

1. Open up the IoT project and add a new Azure IoT Edge module:



2. Select **Python Module**, name it **CameraModule** and fill up your Azure Container Registry address as you dit for the **ObjectDetection** module.

   You should then have a brand new **CameraModule** folder added to your directory. Note that the *deployment.template.json* file has been updated too - you will see in the next section how to manage the deployment of your modules -.



3. Just like any other module, create an *app/* and *build/* folder to separate the application from the build. Move the *requirements.txt* file to the *build/* folder and name it "arm32v7-requirements.txt". Also move the *main.py* to *app/* folder.

4. As your module requires to have access to the Pi Camera, your container image needs to import the python module *picamera*. Hence, update the *arm32v7-requirements.txt* file:

```
azure-iothub-device-client
numpy
requests
picamera
```

   The `azure-iothub-device-client` module contains the functionalities to define all the connections between the modules and the Hub, to set callbacks for messages and alerts to be sent to the Hub.

5. Create a new *PiCamStream.py* Python script. You need to manage the data stream sent by the Pi Camera to be processed inside the AI module:

```
import io
import time
import threading
import json
import requests

class ImageProcessor(threading.Thread):
    def __init__(self, owner):
        super(ImageProcessor, self).__init__()
        self.stream = io.BytesIO()
        self.event = threading.Event()
        self.terminated = False
        self.owner = owner
        self.start()

    def sendFrameForProcessing(self):
        """send a POST request to the AI module server"""
        headers = {'Content-Type': 'application/octet-stream'}
        try:
            self.stream.seek(0)
            response = requests.post(self.owner.endPointForProcessing, headers = headers,
                                     data = self.stream)
        except Exception as e:
            print('sendFrameForProcessing Exception -' + str(e))
            return "[]"

        return json.dumps(response.json())

    def run(self):
        # This method runs in a separate thread
        while not self.terminated:
            # Wait for an image to be written to the stream
            if self.event.wait(1):
                try:
                    result = self.sendFrameForProcessing()
                    if result != '[]' and self.owner.sendToHubCallback is not None:
                    # send messages to the hub if and only if an object has been detected
                        self.owner.sendToHubCallback(result)
                    #print(result)
                    #self.owner.done=True
```

```
                    # uncomment above if you want the process to terminate under certain conditions
                finally:
                    # Reset the stream and event
                    self.stream.seek(0)
                    self.stream.truncate()
                    self.event.clear()
                    # Return ourselves to the available pool
                    with self.owner.lock:
                        self.owner.pool.append(self)

class ProcessOutput(object):
    def __init__(self, endPoint, functionCallBack):
        self.done = False
        # Construct a pool of 4 image processors along with a lock
        # to control access between threads
        # Note that you can vary the number depending on how many processors your device has
        self.lock = threading.Lock()
        self.endPointForProcessing = endPoint
        self.sendToHubCallback = functionCallBack
        self.pool = [ImageProcessor(self) for i in range(4)]
        self.processor = None

    def write(self, buf):
        if buf.startswith(b'\xff\xd8'):
            # New frame; set the current processor going and grab
            # a spare one
            if self.processor:
                self.processor.event.set()
            with self.lock:
                if self.pool:
                    self.processor = self.pool.pop()
                else:
                    # No processor's available, we'll have to skip
                    # this frame; you may want to print a warning
                    # here to see whether you hit this case
                    self.processor = None
        if self.processor:
            self.processor.stream.write(buf)

    def flush(self):
        # When told to flush (this indicates end of recording), shut
        # down in an orderly fashion. First, add the current processor
        # back to the pool
        if self.processor:
            with self.lock:
                self.pool.append(self.processor)
                self.processor = None
        # Now, empty the pool, joining each thread as we go
        while True:
            with self.lock:
                try:
                    proc = self.pool.pop()
                except IndexError:
                    pass # pool is empty
            proc.terminated = True
            proc.join()
```

You want to capture and process images at the same time and within the same script. With the class `threading`, you write a subclass `ImageProcessor` that defines the thread that will send images to the

REST API so that you can set up a pool of parallel `ImageProcessor` threads to accept and process image streams as captures come in. The number of threads will depend on the device you are working on.

6. Take a look now at the *main.py* script. As the device will send a message to the hub every time there is an object detected. You need to configure what we call the `HubManager`:

```
class HubManager(object):

    def __init__(
            self,
            protocol=IoTHubTransportProvider.MQTT):
        self.client_protocol = protocol
        self.client = IoTHubModuleClient()
        self.client.create_from_environment(protocol)

        # set the time until a message times out
        self.client.set_option("messageTimeout", MESSAGE_TIMEOUT)

        # sets the callback when a message arrives on "input1" queue.  Messages sent to
        # other inputs or to the default will be silently discarded.
        self.client.set_message_callback("input1", receive_message_callback, self)

    # Forwards the message received onto the next stage in the process.
    def forward_event_to_output(self, outputQueueName, event, send_context):
        self.client.send_event_async(
            outputQueueName, event, send_confirmation_callback, send_context)
```

In the given example, every time a message arrives on the input queue, it calls the function `receive_message_callback` which consists in incrementing the number of callbacks and discard the input shortly after. The method `forward_event_to_output` simply forwards the message to the `outputQueue` of the module.

In your project, you will only send alerts to the hub if the prediction is different than `'[]'` (with `send_to_Hub_callback`) and you will be forwarding message received to the "`output1`" queue as well, *main.py* looks like this:

```
import random
import time
import sys

# use env variables
import os

import iothub_client
# pylint: disable=E0611
from iothub_client import IoTHubModuleClient, IoTHubClientError, IoTHubTransportProvider
from iothub_client import IoTHubMessage, IoTHubMessageDispositionResult, IoTHubError

# picamera imports
import PiCamStream
from PiCamStream import ProcessOutput
from picamera import PiCamera

# messageTimeout - the maximum time in milliseconds until a message times out.
# The timeout period starts at IoTHubModuleClient.send_event_async.
# By default, messages do not expire.
MESSAGE_TIMEOUT = 10000
```

```
# global counters
SEND_CALLBACKS = 0

# Choose HTTP, AMQP or MQTT as transport protocol.  Currently only MQTT is supported.
PROTOCOL = IoTHubTransportProvider.MQTT

# Send message to the Hub and forwards to the "output1" queue
def send_to_Hub_callback(strMessage):
    message = IoTHubMessage(bytearray(strMessage, 'utf8'))
    hubManager.send_event_to_output("output1", message, 0)

# Callback received when the message that we're forwarding is processed
def send_confirmation_callback(message, result, user_context):
    global SEND_CALLBACKS
    print ( "Confirmation[%d] received for message with result = %s" % (user_context, result) )
    map_properties = message.properties()
    key_value_pair = map_properties.get_internals()
    print ( "    Properties: %s" % key_value_pair )
    SEND_CALLBACKS += 1
    print ( "    Total calls confirmed: %d" % SEND_CALLBACKS )

# Set up the HubManager
class HubManager(object):

    def __init__(
            self,
            messageTimeout,
            protocol):
        self.client_protocol = protocol
        self.client = IoTHubModuleClient()
        self.client.create_from_environment(protocol)

        # set the time until a message times out
        self.client.set_option("messageTimeout", messageTimeout)

    # Forwards the message received onto the next stage in the process.
    def send_event_to_output(self, outputQueueName, event, send_context):
        self.client.send_event_async(
            outputQueueName, event, send_confirmation_callback, send_context)

def main(messageTimeout, protocol, imageProcessingEndpoint=""):
    try:
        print ( "\nPython %s\n" % sys.version )
        print ( "PiCamera module running. Press CTRL+C to exit" )
        try:
            global hubManager
            hubManager = HubManager(messageTimeout, protocol)
        except IoTHubError as iothub_error:
            print ( "Unexpected error %s from IoTHub" % iothub_error )
            return
        with PiCamera(resolution='VGA') as camera:
            camera.start_preview()
            time.sleep(2)
            output = ProcessOutput(imageProcessingEndpoint, send_to_Hub_callback)
            camera.start_recording(output, format='mjpeg')
            while not output.done:
                camera.wait_recording(1)
            camera.stop_recording()
    except KeyboardInterrupt:
        print ("PiCamera module stopped")
```

```
if __name__ == '__main__':
    try:
        IMAGE_PROCESSING_ENDPOINT = os.getenv('IMAGE_PROCESSING_ENDPOINT', "")
    except ValueError as error:
        print ( error )
        sys.exit(1)

    main(MESSAGE_TIMEOUT, PROTOCOL, IMAGE_PROCESSING_ENDPOINT)
```

7. Write the Docker file (*Dockerfile.arm32v7*) to configure how the module will be running.

```
FROM balenalib/raspberrypi3:stretch
# The balena base image for building apps on Raspberry Pi 3.

RUN echo "BUILD MODULE: PiCamera"

# Enforces cross-compilation through Quemu
RUN [ "cross-build-start" ]

# Update package index and install dependencies
RUN install_packages \
    python3 \
    python3-pip \
    python3-dev \
    build-essential \
    libopenjp2-7-dev \
    zlib1g-dev \
    libatlas-base-dev \
    wget \
    libboost-python1.62.0 \
    curl \
    libcurl4-openssl-dev

# Install Python packages
COPY /build/arm32v7-requirements.txt ./
RUN pip3 install --upgrade pip
RUN pip3 install --upgrade setuptools
RUN pip3 install --index-url=https://www.piwheels.org/simple -r arm32v7-requirements.txt

# Cleanup
RUN rm -rf /var/lib/apt/lists/* \
    && apt-get -y autoremove

RUN [ "cross-build-end" ]

ADD /app/ .

ENTRYPOINT [ "python3", "-u", "./main.py" ]
```

Module 3: Deploying the Azure IoT Edge modules

Overview

Now that you have your Azure IoT Edge modules ready, i.e. your tailored for purpose IoT solution, the last step is to deploy them and run them on your Azure IoT Edge device.

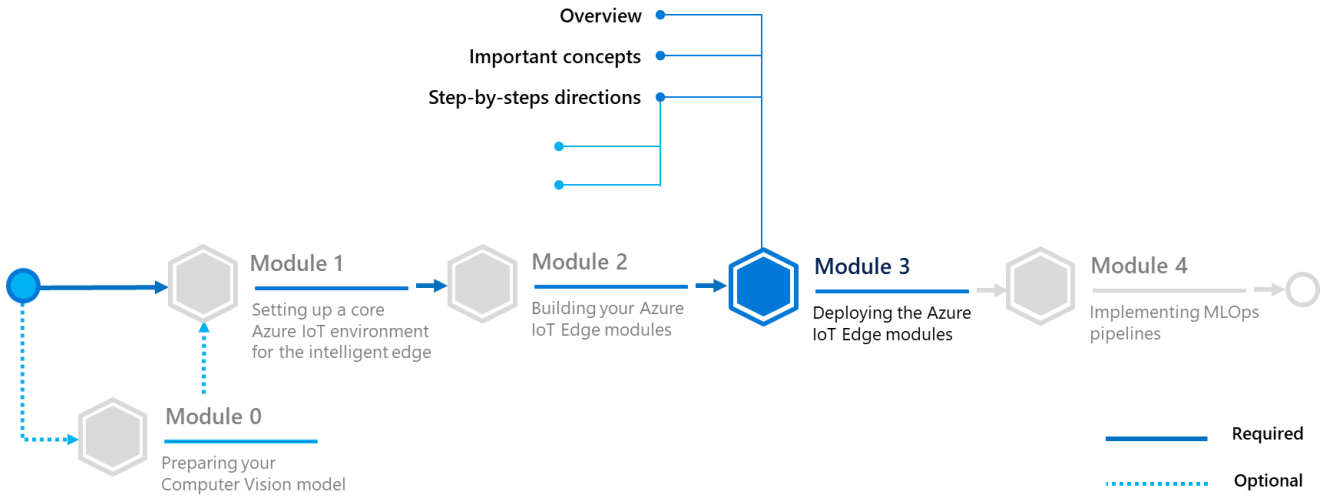
Once deployed, you will no longer need to tell edge devices what they have to do so you have to specify beforehand how each module interact with each other and when to report data to the hub (the advantage is to give workloads to the edge so that only important events are reported).

Azure IoT Edge hub facilitates module to module communication. Using the IoT 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.

For that, you need to write a deployment manifest that will contain all the informations for the IoT solution to work.

In this module, you will learn to:

- Write a deployment manifest.
- Deploy the IoT solution to the edge.
- Monitor in-and-out messages with your Azure IoT Hub.



Before dive in, let's consider some important concepts notably regarding the structure of an Azure IoT Edge device.

Important concepts

Structure of an Azure IoT Edge device

Each Azure IoT Edge device runs at least two modules: `$edgeAgent` and `$edgeHub`, which are part of the Azure IoT Edge runtime as you saw. Azure IoT Edge device can run multiple additional modules for any number of processes.

The deployment manifest is a JSON document that describes:

- The Azure IoT Edge agent module twin, which includes three components.
    1. The container image for each module that runs on the device.
    2. The credentials to access private container registries that contain module images (optional if working wih public registry).
    3. Instructions for how each module should be created and managed.
- The Azure IoT Edge hub module twin, which includes how messages flow between modules and eventually to Azure IoT Hub.

Optionally, the desired properties of any additional module twins.

> **Note** A module twin is simply a piece of JSON document stored in Azure IoT Hub, that contains state information for a module instance, including metadata, configurations, and conditions.

A module twin includes:

- **Desired properties** that are set when applying a deployment manifest on a specific device as part of a single-device or at-scale deployment.
- **Reported properties**: that reports the status of the application of the last-seen desired properties, the status of the modules currently running on the device, as reported by the Azure IoT Edge agent and has a copy of the desired properties currently running on the device. This has the use to verify if the current desired properties are indeed the properties defined in the latest deployment manifest.

An Azure IoT Edge device, in addition to the two required modules above, can contain up to 20 custom modules. Every deployment manifest follows this structure:

```
{
    "modulesContent": {
        "$edgeAgent": { // required
            "properties.desired": {
                // desired properties of the Edge agent
                // includes the image URIs of all modules
                // includes container registry credentials
            }
        },
        "$edgeHub": { //required
            "properties.desired": {
                // desired properties of the Edge hub
                // includes the routing information between modules, and to IoT Hub
            }
        },
        "module1": {  // optional
```

```
            "properties.desired": {
                // desired properties of module1
            }
        },
        "module2": {  // optional
            "properties.desired": {
                // desired properties of module2
            }
        },
        ...
    }
}
```

The module twins `$edgeAgent` and `$edgeHub` are required and some of its desired properties are required too. Check out here[83] for an exhaustive list of properties of both module twins.

Let's see how to configure both modules.

Configuration of the edge Agent

The edge Agent is the runtime component that manages installation, updates, and status reporting for an Azure IoT Edge device. Therefore, the `$edgeAgent` module twin requires the configuration and management information for all modules. The `$edgeAgent` needs to be configured as follows:

```
"$edgeAgent": {
  "properties.desired": {
    "schemaVersion": "1.0",
    "runtime": {
      "type": "docker",
      "settings": {
        "minDockerVersion": "v1.25", // specify the minimum accepted version of docker
        "registryCredentials": {
          // (Optional) give access to private container registry
          }
        }
      }
    },
    "systemModules": {
      "edgeAgent": {
        "type": "docker",
        "settings": {
          "image": "mcr.microsoft.com/azureiotedge-agent:1.0",
          "createOptions": ""
        }
      },
      "edgeHub": {
        "type": "docker",
        "status": "running",
        "restartPolicy": "always",
        "settings": {
```

[83] PROPERTIES OF THE IOT EDGE AGENT AND IOT EDGE HUB MODULE TWINS: https://docs.microsoft.com/en-us/azure/iot-edge/module-edgeagent-edgehub

```
          "image": "mcr.microsoft.com/azureiotedge-hub:1.0",
          "createOptions": ""
        }
      }
    },
    "modules": {
      "module1": {
        // configuration of module1
      },
      "module2": {
        // configuration of module2
      }
    }
  }
}
```

Let's break down the desired properties of the `$edgeAgent` that are required (there is a more exhaustive list of all the properties above):

Property	Requirement
schemaVersion	Has to be "1.0"
runtime.type	Has to be "docker"
runtime.settings.minDockerVersion	Set to the minimum Docker version required by this deployment manifest
systemModules.edgeAgent.type	Has to be "docker"
systemModules.edgeAgent.settings.image	The URL of the image of the Edge agent (mcr.microsoft.com/azureiotedge-agent:1.0)
systemModules.edgeHub.type	Has to be "docker"
systemModules.edgeHub.status	Has to be "running"
systemModules.edgeHub.restartPolicy	Has to be "always"
systemModules.edgeHub.settings.image	The URL of the image of the Edge agent (mcr.microsoft.com/azureiotedge-hub:1.0)

The modules configured by the $edgeAgent require the following properties:

Property	Requirement
modules.{moduleId}.type	Has to be "docker"
modules.{moduleId}.status	Has to be either "running" or "stopped"
modules.{moduleId}.restartPolicy	Has to be either "never", "on-failure", "on-unhealthy" or "always"
modules.{moduleId}.settings.image	The URL of the docker image of the module

Note For each module and systemModule above, there is a parameter called *createOptions* that enable you to pass on arguments into your docker image when run. Whether it is to expose a port or to set up the environment, you can pass on variables to customize the use of your module. For more information, see article CREATE A CONTAINER[84].

Configuration the edge Hub

The edge Hub is responsible for managing all the routes between the modules. Thus, it contains a desired property "routes" which is configured as follows:

```
"$edgeHub": {
    "properties.desired": {
        "routes": {
            "route1": "FROM <source> WHERE <condition> INTO <sink>",
            "route2": "FROM <source> WHERE <condition> INTO <sink>"
            "route3": "FROM <source> WHERE <condition> INTO <sink>"
            "route4": …
        },
    }
}
```

Every route needs a `<source>` and a `<sink>`, the condition is optional and used to design much more complex solutions.

Note For more information, see section Condition[85] in article LEARN HOW TO DEPLOY MODULES AND ESTABLISH ROUTES IN IOT EDGE.

Source	Description
*/**	All device-to-cloud messages or twin change notifications from any module or leaf device
/twinChangeNotifications	Any twin change (reported properties) coming from any module or leaf device
*/messages/**	Any device-to-cloud message sent by a module through some or no output, or by a leaf device
*/messages/modules/**	Any device-to-cloud message sent by a module through some or no output
*/messages/modules/<moduleId>/**	Any device-to-cloud message sent by a specific module <moduleId> through some or no output
*/messages/modules/<moduleId>/outputs/**	Any device-to-cloud message sent by a specific module <moduleId> through some output

[84] CREATE A CONTAINER: https://docs.docker.com/engine/api/v1.32/#operation/ContainerCreate

[85] LEARN HOW TO DEPLOY MODULES AND ESTABLISH ROUTES IN IOT EDGE: https://docs.microsoft.com/en-us/azure/iot-edge/module-composition#condition

/messages/modules/<moduleId>/outputs/<output>	Any device-to-cloud message sent by a specific module <moduleId> through a specific output <outpout>

Sink	Description
$upstream	Send the message to Azure IoT Hub
BrokeredEndpoint("/modules/<moduleId>/inputs/<input>")	Send the message to <input> of module <moduleId>

The Azure IoT Edge hub stores the messages locally in case a route can't deliver the message to its sink. There is a required edge hub desired property `storeAndForwardConfiguration.timeToLiveSecs` that decides up to when the messages are stored before being deleted.

Step-by-step directions

Based on the above understanding, this module covers the following two activities:

1. Writing your deployment manifest.
2. Building and deploying the solution.

Each activity is described in order in the next sections.

Writing your deployment manifest

The steps in this section might be typically performed by device developers and cloud developers.

Let's illustrate what you have seen above by writing your own deployment manifest. So far, the *deployment.template.json* file should look like this (lines that are highlighted in yellow create a simulated sensor module that need to be removed since it has been automatically generated by default by Visual Studio Code):

```
{
  "$schema-template": "2.0.0",
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": "",
            "registryCredentials": {
              "YOUR_REGISTRY_NAME": {
                "username": "$CONTAINER_REGISTRY_USERNAME",
                "password": "$CONTAINER_REGISTRY_PASSWORD",
                "address": "YOUR_REGISTRY_NAME.azurecr.io"
              }
            }
          }
        },
        "systemModules": {
```

```
          "edgeAgent": {
            "type": "docker",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-agent:1.0",
              "createOptions": {}
            }
          },
          "edgeHub": {
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-hub:1.0",
              "createOptions": {
                "HostConfig": {
                  "PortBindings": {
                    "5671/tcp": [
                      {
                        "HostPort": "5671"
                      }
                    ],
                    "8883/tcp": [
                      {
                        "HostPort": "8883"
                      }
                    ],
                    "443/tcp": [
                      {
                        "HostPort": "443"
                      }
                    ]
                  }
                }
              }
            }
          }
        },
        "modules": {
          "ObjectDetection": {
            "version": "1.0",
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "${MODULES.ObjectDetection}",
              "createOptions": {}
            }
          },
          "SimulatedTemperatureSensor": {
            "version": "1.0",
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-simulated-temperature-sensor:1.0",
              "createOptions": {}
            }
          },
          "CameraModule": {
            "version": "1.0",
            "type": "docker",
```

```
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "${MODULES.CameraModule}",
              "createOptions": {}
            }
          }
        }
      }
    },
    "$edgeHub": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "routes": {
          "ObjectDetectionToIoTHub": "FROM /messages/modules/ObjectDetection/outputs/* INTO
$upstream",
          "CameraModuleToIoTHub": "FROM /messages/modules/CameraModule/outputs/* INTO $upstream"
        },
        "storeAndForwardConfiguration": {
          "timeToLiveSecs": 7200
        }
      }
    }
  }
}
```

The port bindings defined in the edge Hub are here to ensure proper network and firewall rules to set up for secure edge to cloud communication as Azure IoT Edge is still dependent on the underlying machine and network configuration. Hence, port 8883 is reserved for protocol MQTT, 5671 for AMQT and 443 for HTTPS.

Let's now consider the configuration that pertains to the Vision AI Developer Kit.

Using a Raspberry Pi 3

Configuring the edge Agent for your solution

The Azure IoT Edge hub, which is part of the Azure IoT Edge runtime, is optimized for performance by default and attempts to allocate large chunks of memory. This optimization is not ideal for constrained edge devices like the Raspberry Pi 3 and can cause stability problems.

Thus, in `systemModules.edgeHub`, add an environment variable called `OptimizeForPerformance` as follows:

```
"edgeHub": {
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-hub:1.0",
              "createOptions": {
                "HostConfig": {
                  "PortBindings": {
                    "5671/tcp": [
                      {
                        "HostPort": "5671"
                      }
                    ],
```

```
                    "8883/tcp": [
                      {
                        "HostPort": "8883"
                      }
                    ],
                    "443/tcp": [
                      {
                        "HostPort": "443"
                      }
                    ]
                  }
                }
              }
            },
            "env": {
              "OptimizeForPerformance": {
                "value": "false"
              }
            }
          }
```

If you recall, in your *main.py* script, the arguments of of main function reads the environment variable `IMAGE_PROCESSING_ENDPOINT`. To pass the address of your REST server, add an environment variable in `modules.CameraModule`:

```
"env": {
              "IMAGE_PROCESSING_ENDPOINT": {
                "value":"http://ObjectDetection:8885/image"
                }
              }
```

For your `ObjectDetection` module, you need to bind the port 8885 for processing, to do that, you need to specify the port binding in `modules.ObjectDetection.createOptions`:

```
"createOptions": {
                "HostConfig": {
                  "PortBindings": {
                    "8000/tcp": [
                      {
                        "HostPort": "8885"
                      }
                    ]
                  }
                }
              }
```

For your `CameraModule`, you need to allow the container to access the picamera device for that, you need to specify the device inside (here the device is */dev/vchiq* and */dev/vcsm*). To do that, in `modules.CameraModule.createOptions`:

```
"HostConfig":{
                  "Binds":["/dev/vchiq:/dev/vchiq", "/dev/vcsm:/dev/vcsm"],
                  "Devices":[{"PathOnHost":"/dev/vchiq","PathInContainer":"/dev/vchiq","CgroupPermiss
ions":"mrw"},{"PathOnHost":"/dev/vcsm","PathInContainer":"/dev/vcsm","CgroupPermissions":"mrw"}]
                }
```

Configuring the edge Hub for your solution

As your `ObjectDetection` module only sets up a REST API locally inside of your RPI3, it will not interact with the other modules nor send data to the hub. Hence, you can delete the following route:

```
"ObjectDetectionToIoTHub": "FROM /messages/modules/ObjectDetection/outputs/* INTO $upstream"
```

As you forward the message to the "`output1`" queue, specify it :

```
"CameraModuleToIoTHub": "FROM /messages/modules/camera-module/outputs/output1 INTO $upstream"
```

Thus, the deployment manifest finally looks like this :

```
{
  "$schema-template": "2.0.0",
  "modulesContent": {
    "$edgeAgent": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "runtime": {
          "type": "docker",
          "settings": {
            "minDockerVersion": "v1.25",
            "loggingOptions": "",
            "registryCredentials": {
              "YOUR_REGISTRY_NAME": {
                "username": "$CONTAINER_REGISTRY_USERNAME ",
                "password": "$CONTAINER_REGISTRY_PASSWORD ",
                "address": "YOUR_REGISTRY_NAME.azurecr.io"
              }
            }
          }
        },
        "systemModules": {
          "edgeAgent": {
            "type": "docker",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-agent:1.0",
              "createOptions": {}
            }
          },
          "edgeHub": {
            "type": "docker",
            "status": "running",
            "restartPolicy": "always",
            "settings": {
              "image": "mcr.microsoft.com/azureiotedge-hub:1.0",
              "createOptions": {
                "HostConfig": {
                  "PortBindings": {
                    "5671/tcp": [
                      {
                        "HostPort": "5671"
```

```
                    }
                  ],
                  "8883/tcp": [
                    {
                      "HostPort": "8883"
                    }
                  ],
                  "443/tcp": [
                    {
                      "HostPort": "443"
                    }
                  ]
                }
              }
            }
          },
          "env": {
            "OptimizeForPerformance": {
              "value": "false"
            }
          }
        }
      },
      "modules": {
        "object-detection": {
          "version": "1.0",
          "type": "docker",
          "status": "running",
          "restartPolicy": "always",
          "settings": {
            "image": "${MODULES.ObjectDetection}",
            "createOptions": {
              "HostConfig": {
                "PortBindings": {
                  "8000/tcp": [
                    {
                      "HostPort": "8885"
                    }
                  ]
                }
              }
            }
          }
        },
        "camera-module": {
          "version": "1.0",
          "type": "docker",
          "status": "running",
          "restartPolicy": "always",
          "settings": {
            "image": "${MODULES.CameraModule}",
            "createOptions": {
              "HostConfig":{
                "Binds":["/dev/vchiq:/dev/vchiq", "/dev/vcsm:/dev/vcsm"],
                "Devices":[{"PathOnHost":"/dev/vchiq","PathInContainer":"/dev/vchiq","CgroupPermiss
ions":"mrw"},{"PathOnHost":"/dev/vcsm","PathInContainer":"/dev/vcsm","CgroupPermissions":"mrw"}]
              }
            }
          },
          "env": {
            "IMAGE_PROCESSING_ENDPOINT": {
```

```
                "value":"http://object-detection:8885/image"
              }
            }
          }
        }
      }
    },
    "$edgeHub": {
      "properties.desired": {
        "schemaVersion": "1.0",
        "routes": {
          "CameraModuleToIoTHub": "FROM /messages/modules/camera-module/outputs/output1 INTO
$upstream"
        },
        "storeAndForwardConfiguration": {
          "timeToLiveSecs": 7200
        }
      }
    }
  }
}
```

Building and deploying the solution

The steps in this section might be typically performed by device developers and cloud developers.

Now that you have everything ready, you need to build your images.

Perform the following steps:

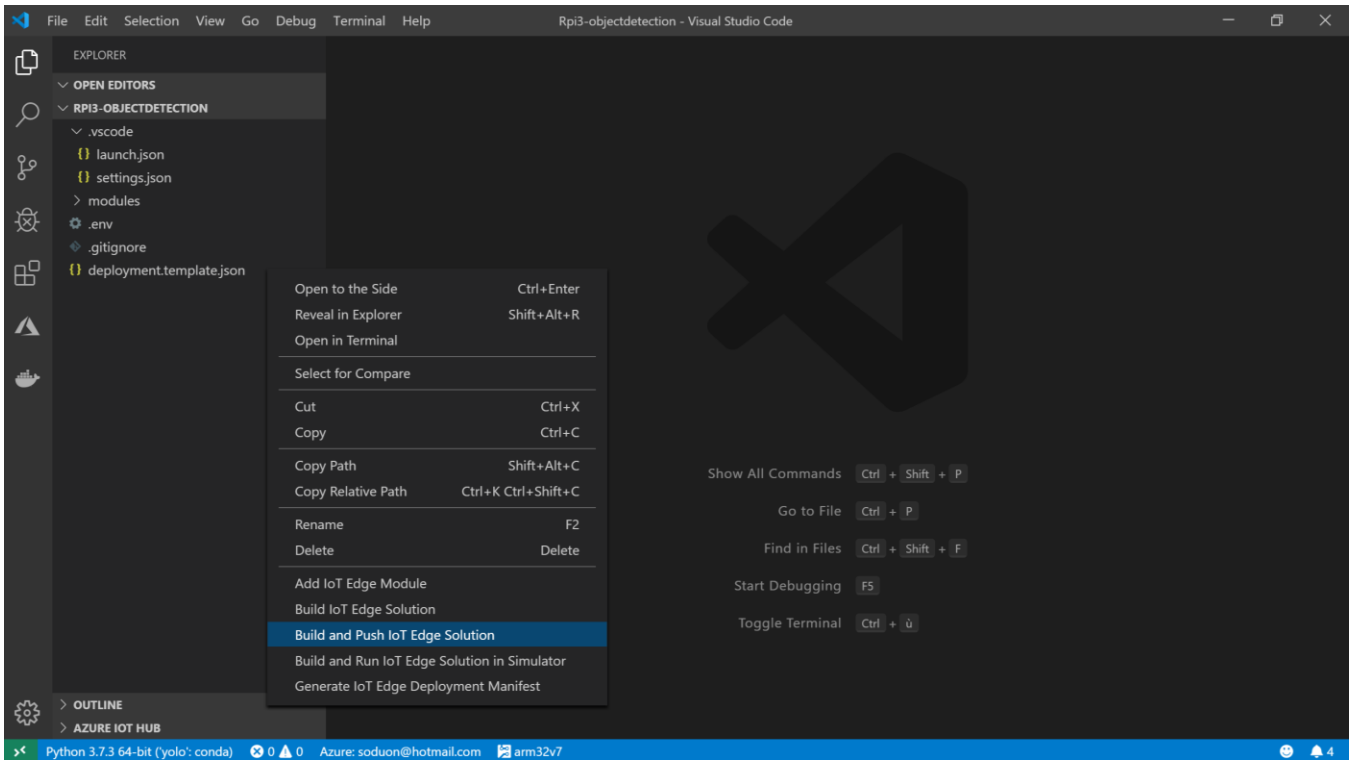
1. As you are using Azure Container Registries here as your private registries, log in to your ACR through the Docker CLI by opening up a terminal in Visual Studio Code :

```
az acr login --name YOUR_REGISTRY_NAME
```

   If you are using your own docker registry, log in with:

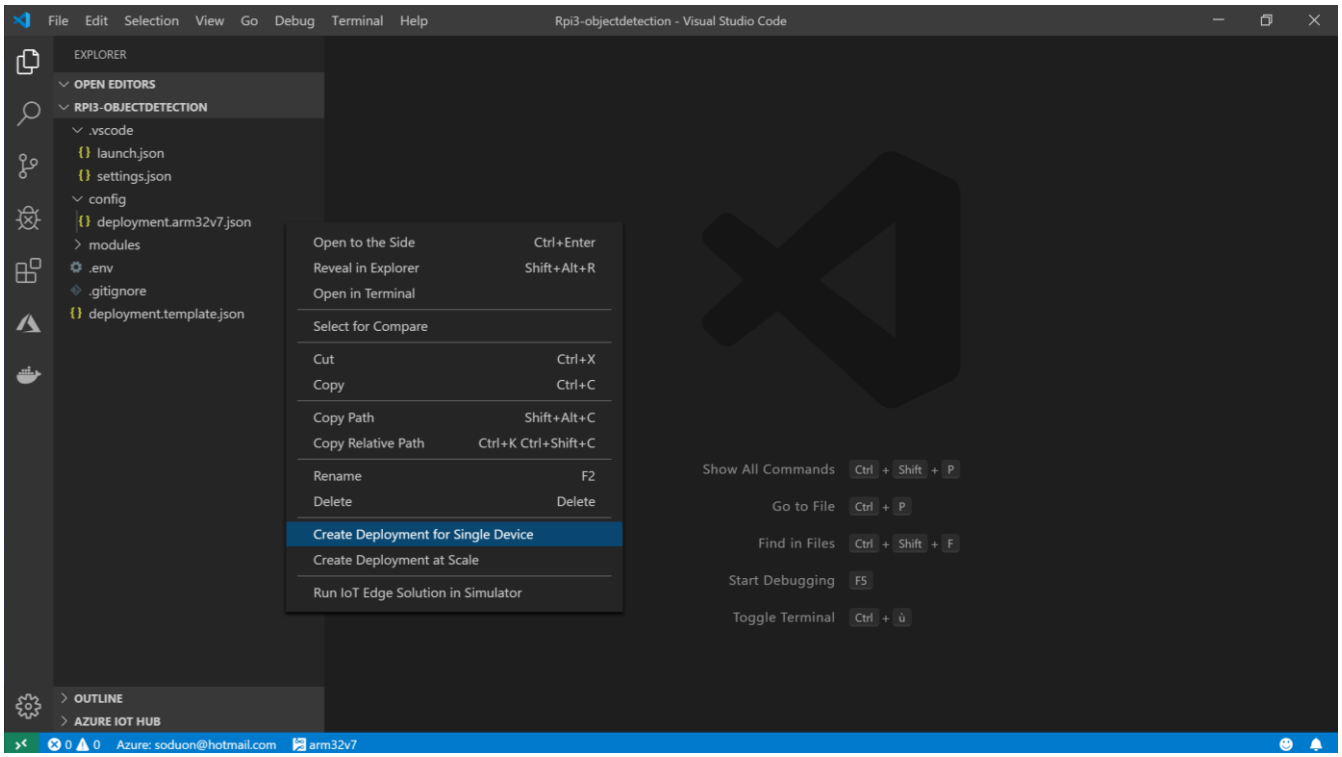
```
docker login –username … -password …
```

2. Once logged in, build and push the image by right-clicking on the *deployment.template.json* file. This will take a couple minutes as it needs to pull the Raspbian stretch image, builds the two modules and pushes them to your private registries:

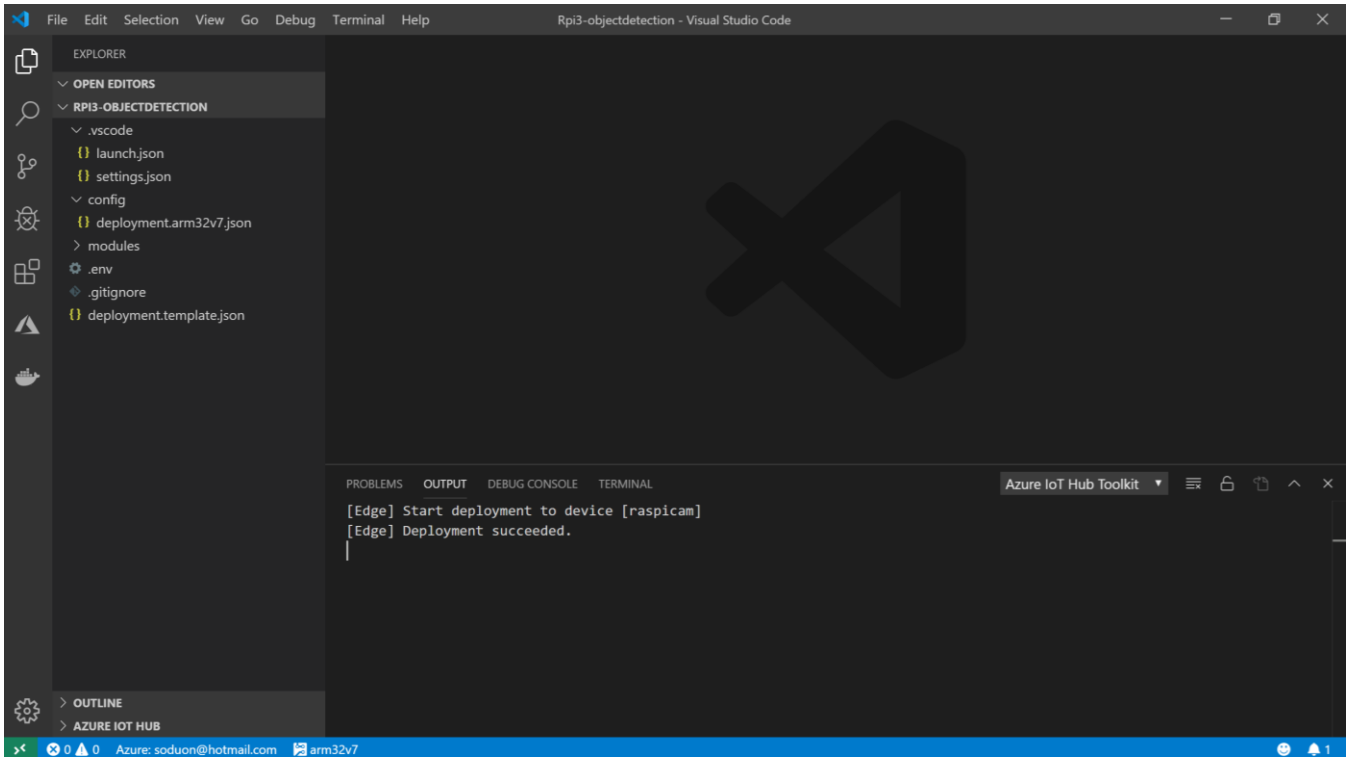


3. Once finished, there should be a new *config* folder created inside of your directory. It contains a *deployment.arm32v6.json* file: this is the final deployment manifest.

   Right-click and select **Deploy for a single device**:



4. Select your Azure IoT Edge device. You have successfully deployed your solution to the edge.

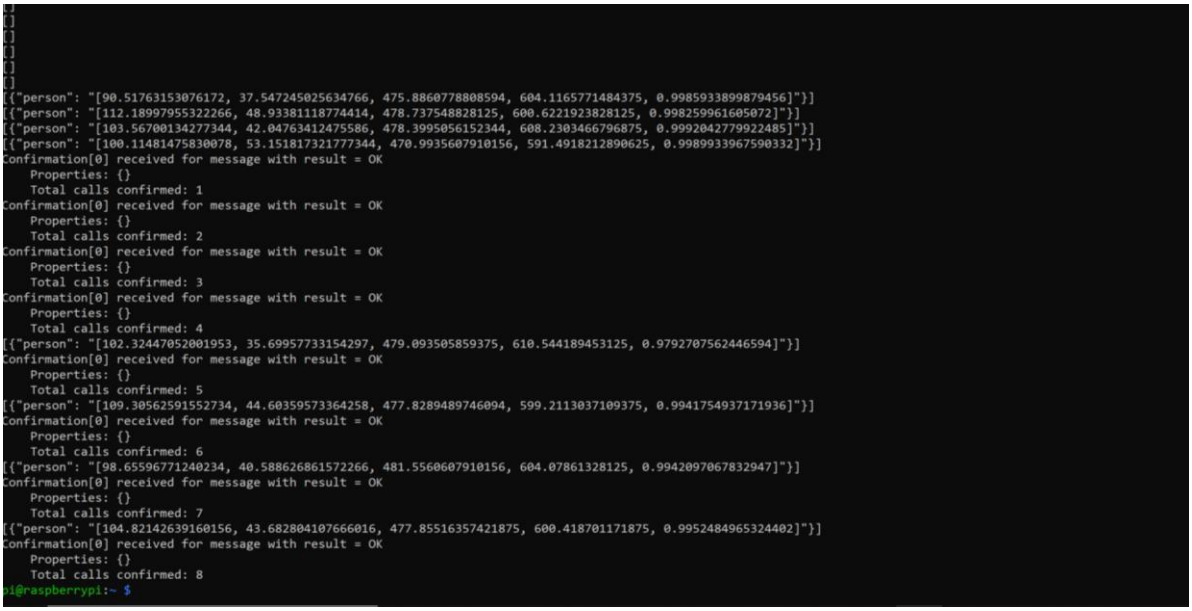


5. Now, ssh into your raspberry pi and type in a terminal:

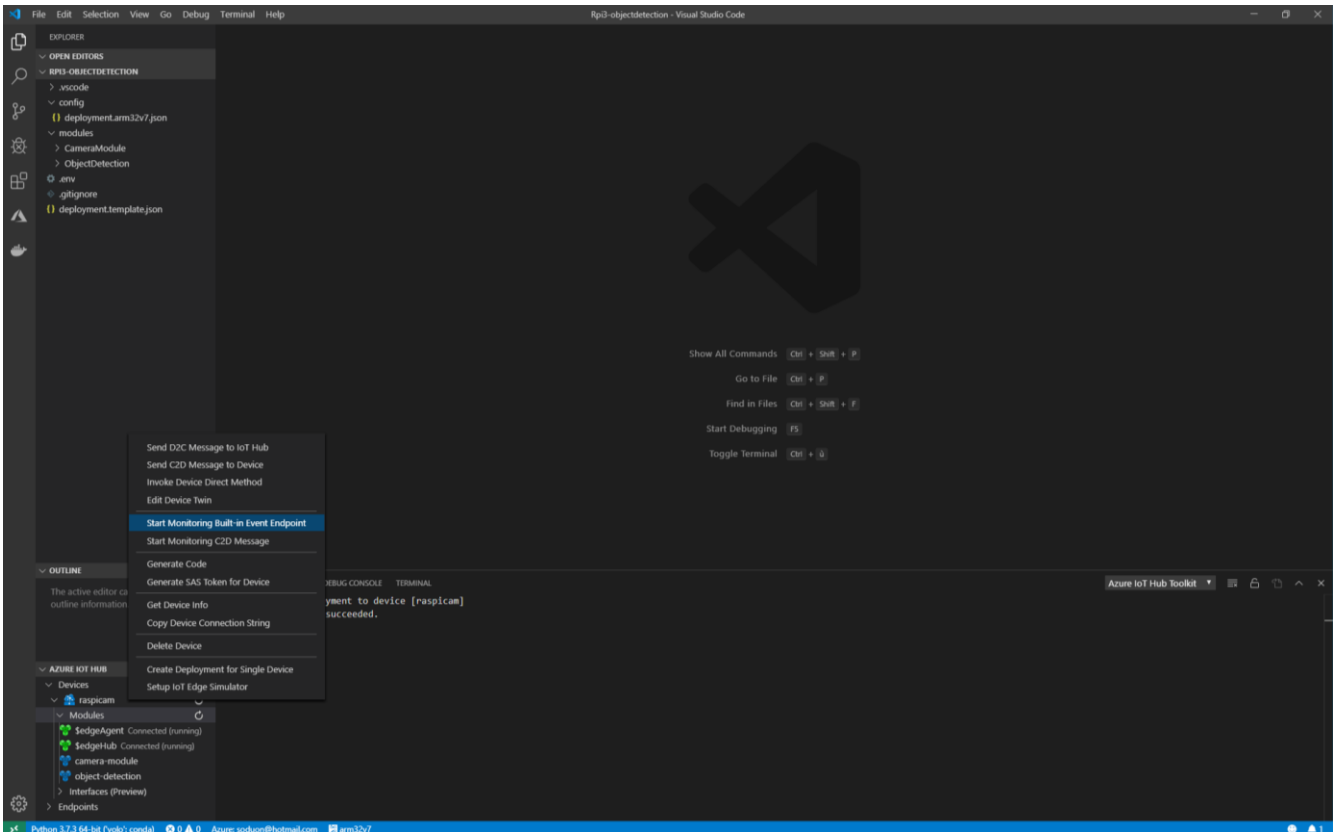
```
iotedge list
```

6. After a few minutes, you should see your modules running and if you look at the logs of your camera module, you should start to see:

```
iotedge logs camera-module
```

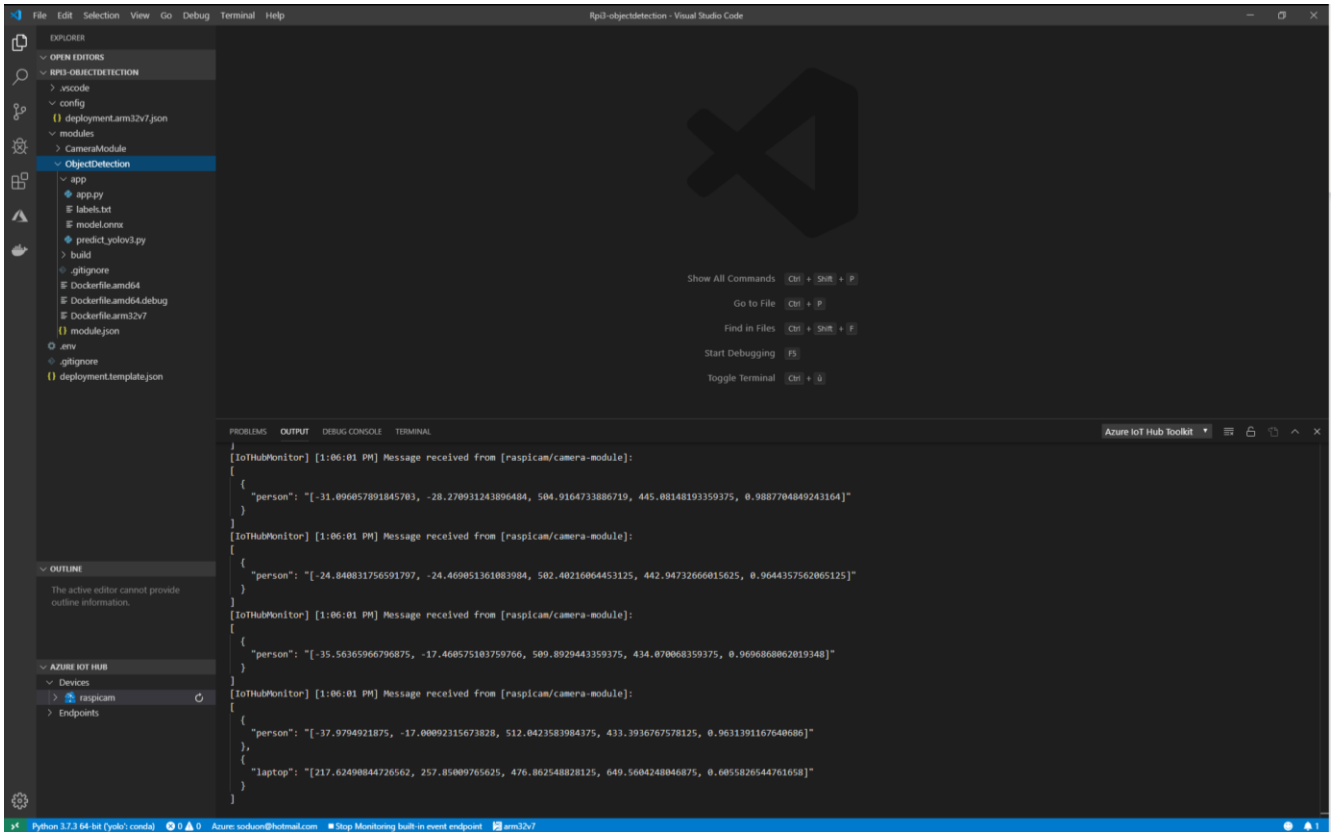


You can also visualize in-and-out messages from Visual Studio Code as follows:



You should start to see after a few seconds the results of the prediction on Visual Studio Code.

Below are predictions you can get by wandering around in your workspace:



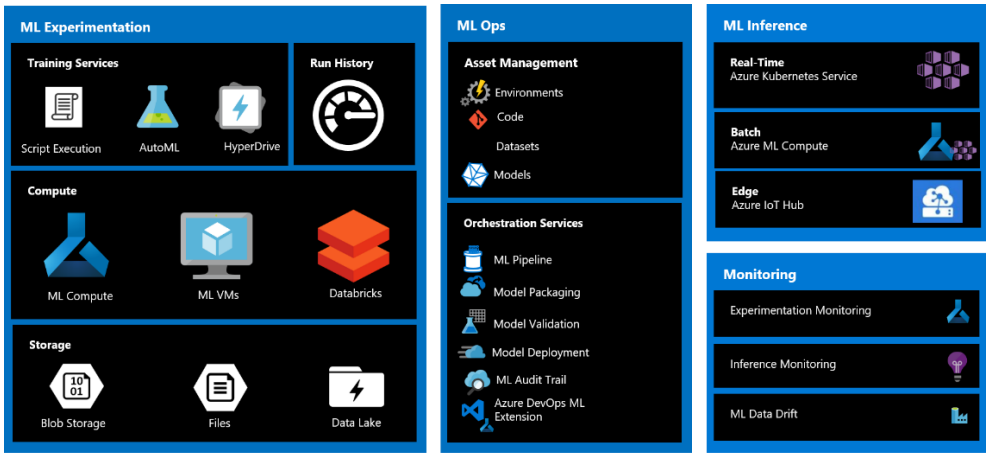
You might notice that this is still very far from real-time predictions since it takes couple seconds to process and display.

Module 4: Implementing MLOps pipelines

Overview

In an effort to bring your IoT solution into production and to simplify future deployments on your devices, you can create via Azure DevOps custom pipelines to configure Continuous Integration/Continuoys Delivery (CI/CD) from commit changes to releases of your project. Equipped with built-in Azure IoT Edge and Azure Machine Learning tasks, Azure Pipelines[86] allows you to easily configure pipelines to train and to deploy newer versions of your model to your IoT devices.

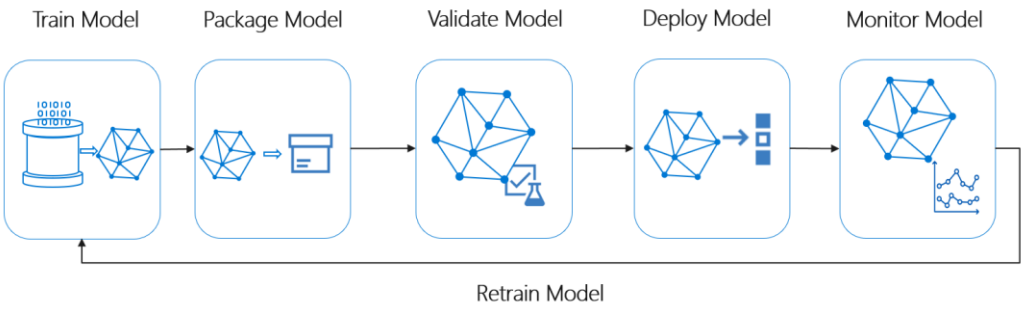
As you saw in the previous modules, Azure does provide you a complete panel of tools to take your model from training to production:



Implementing MLOps means designing pipelines to ease collaboration between data scientists and software engineers, making model versioning and lifecycle management easier for data scientists. By breaking down the process between different release stages, you can evaluate how well the model perfom on local/dev devices before releasing it to an QA environment.

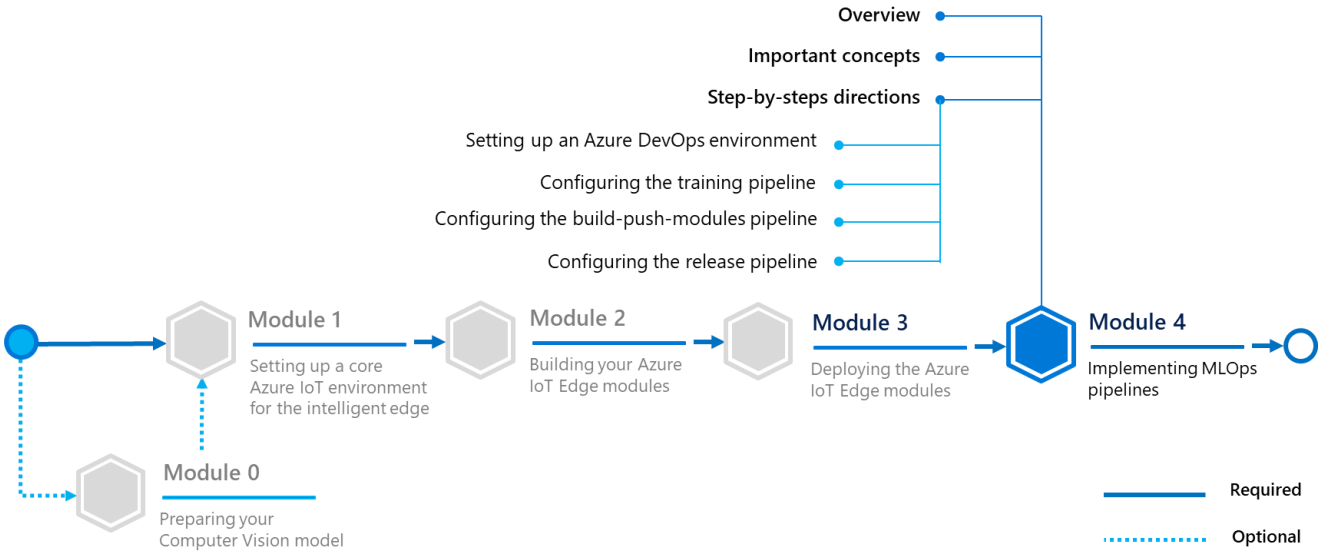
As the accuracy of the model is determined during training and might change over time because of the data drift, determining the value of the trained model and deciding whether or not it should be deployed are crucial to ensure stability and quality.

[86] Azure Pipelines: https://azure.microsoft.com/en-us/services/devops/pipelines/



In this module, you will go again through all the previous modules and build pipelines to connect each step described earlier.

In this module, you will more particulary learn how to set up pipelines to automate the retraining of your YOLOv3 model on the VOC dataset (as you saw in the first module) as well as its deployment on your devices.



For that purpose, the scenario for the current solution is as follows:

1. Data Scientist writes/updates the code, uploads training/testing data if needed and pushes it to the git repo, which triggers the Azure DevOps build pipeline (Continuous Integration).
2. Once the Azure DevOps build pipeline is triggered, it performs code quality checks, data sanity tests, unit tests, builds an Azure ML Pipeline[87] and publishes it in an Azure ML Service Workspace[88].

[87] WHAT ARE AZURE MACHINE LEARNING PIPELINES?: https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-ml-pipelines

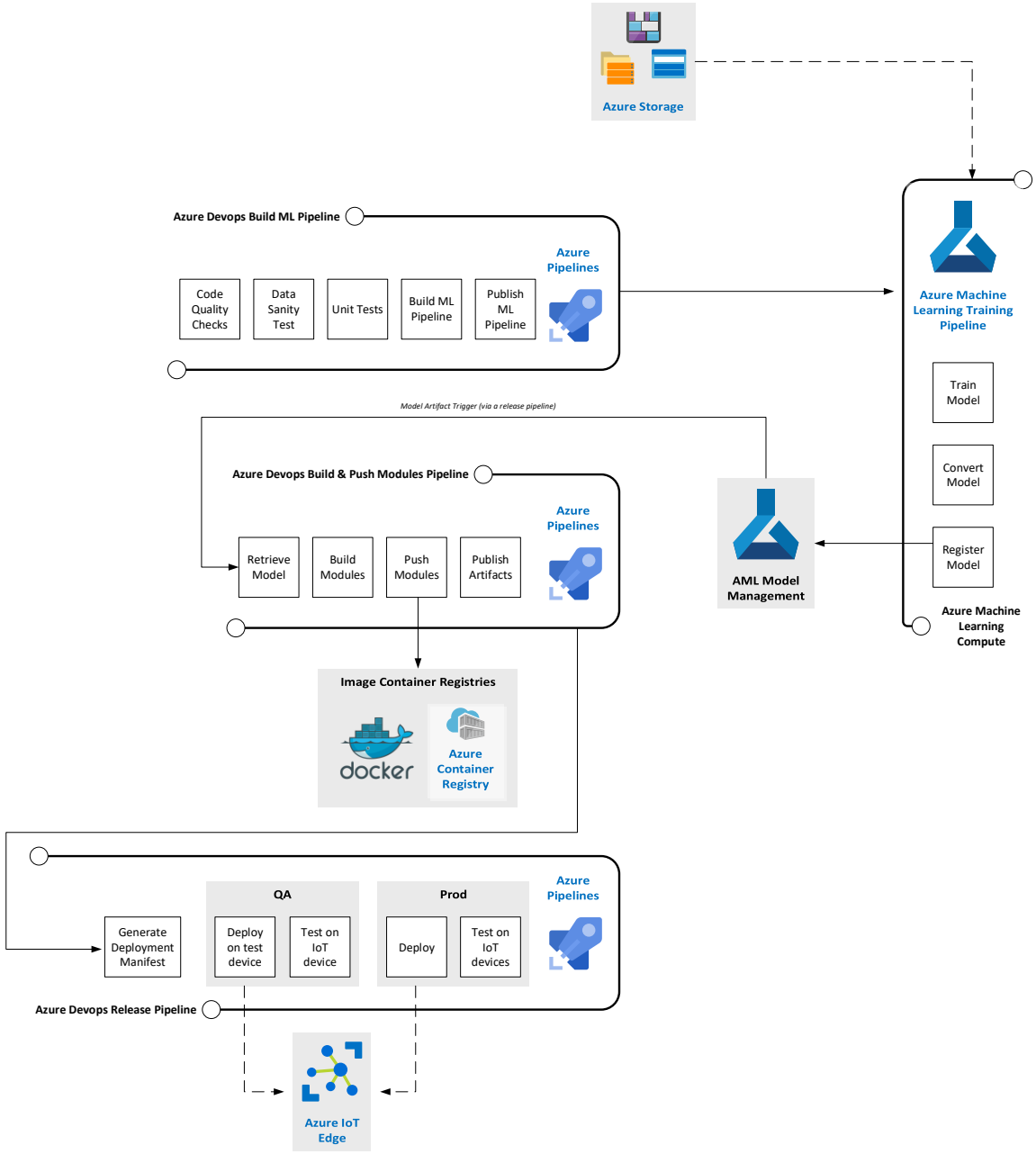
[88] HOW AZURE MACHINE LEARNING WORKS: Architecture and concepts: https://docs.microsoft.com/en-us/azure/machine-learning/service/concept-azure-machine-learning-architecture#workspace

3. The Azure ML Pipeline is triggered once the Azure DevOps Build pipeline completes. All the tasks in this pipeline runs on Azure ML Compute. Following are the tasks in this pipeline:
    - **Train Model** task executes model training script on Azure ML Compute. It outputs a model[89] file which is stored in the run history[90].
    - **Convert Model** task converts the latest model (in .h5) to ONNX format.
    - **Register Model** task takes the converted model and registers it with the Azure ML Model registry[91]. This allows to version control it.
4. Once the ML model is registered on Azure ML, an artifact, that acts as a Release pipeline trigger, will run a Build pipeline containing the IoT solution along with the new trained model, then build and push the containers into your registry.
5. Azure DevOps is packaged with built-in Azure IoT Edge tasks that will enable you to easily generate the manifest and deploy it on your devices via a Release pipeline. You can define your Quality Assurance (QA) environment with a "test device" before deploying your solution in production.

[89] Ibid

[90] Ibid

[91] Ibid



Before diving in, let's consider some important concepts.

Important concepts

Service principals

As you already know, in order to use Azure services, you must first authenticate through the Azure CLI or on the Azure portal. It is by far enough if you want to manage the whole process by yourself (from the training to the deployment on the devices). However, when using automated tools that use Azure services such as Azure

Pipelines, you should always restrict their rights and permissions instead of having applications sign in as a fully privileged user.

For that, Azure offers the possibilities to define service principals that are identities created for use with applications, hosted services, and automated tools to access Azure resources. This access is restricted by the roles assigned to the service principal, giving you control over which resources can be accessed and at which level. For security reasons, it's always recommended to use service principals with automated tools rather than allowing them to log in with a user identity.

Device twins

The notion of device twins has already been mentioned in the last module but to further dive into deployments, you need to understand their role in identifying devices in an IoT Hub.

Device twins are JSON documents that store device state information including metadata, configurations, and conditions. Azure IoT Hub maintains a device twin for each device that you connect to IoT Hub. It consists of :

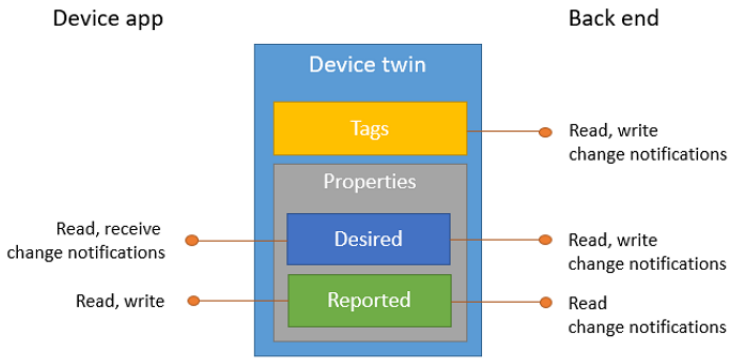
- **Tags**. A section of the JSON document that the solution back end can read from and write to. Tags are not visible to device apps.
- **Desired properties**. Used along with reported properties to synchronize device configuration or conditions. The solution back end can set desired properties, and the device app can read them. The device app can also receive notifications of changes in the desired properties.
- **Reported properties**. Used along with desired properties to synchronize device configuration or conditions. The device app can set reported properties, and the solution back end can read and query them.
- **Device identity properties**. The root of the device twin JSON document contains the read-only properties from the corresponding device identity stored in the identity registry[92].

Whenever, for instance, there is a change in the desired properties, whether it is to add/remove an env variable for a certain module or to pull a newer container image from the registry, the modifications are made to the device twin online. The agent from the IoT Edge Security Daemon[93] then retrieves the module twin on device startup and compares with its actual deployment manifest. Whenever there is a newer version of a module, it then pulls it by inspecting its "twin" on IoT Hub.

That way, device and back ends can synchronize device conditions and configuration. Later on, you will need to set tags to device to separate the different stages (dev, qa, prod).

[92] UNDERSTAND THE IDENTITY REGISTRY IN YOUR IOT HUB: https://docs.microsoft.com/en-us/azure/iot-hub/iot-hub-devguide-identity-registry

[93] AZURE IOT EDGE SECURITY MANAGER: https://docs.microsoft.com/en-us/azure/iot-edge/iot-edge-security-manager



Step-by-step directions

This module covers the following activities:

1. Setting up an Azure DevOps environment.
2. Configuring the training pipeline.
3. Configuring the build-push-modules pipeline.
4. Configuring the release pipeline.

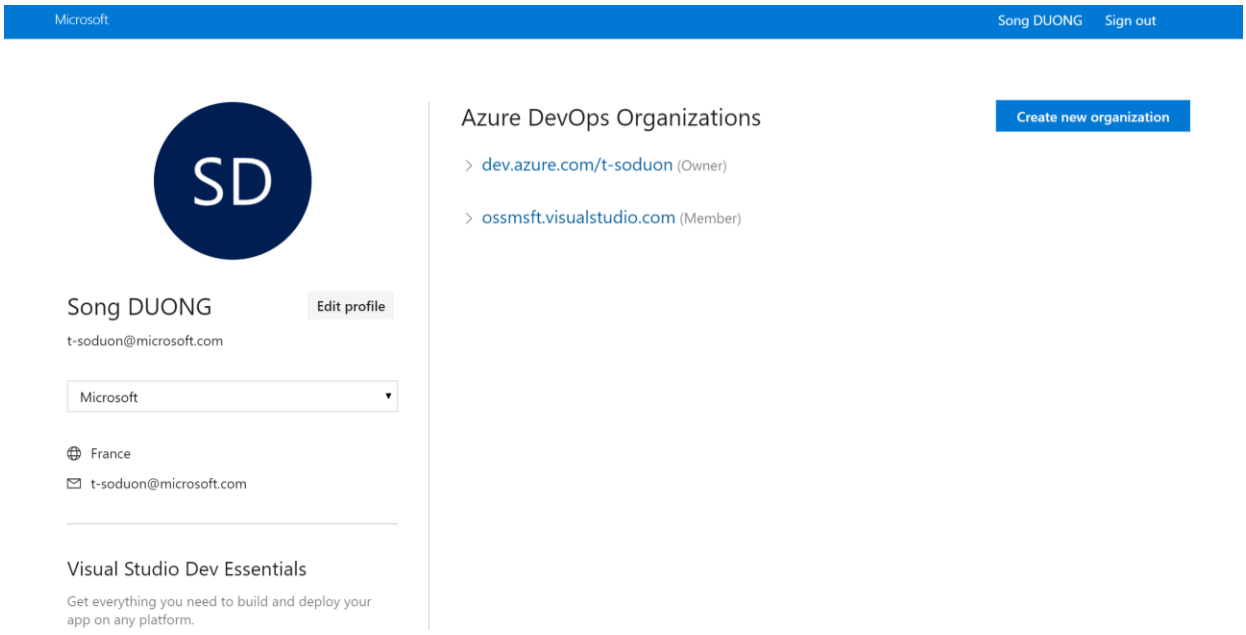
Each activity is described in order in the next sections.

The steps in this section might be typically performed by DevOps engineers.

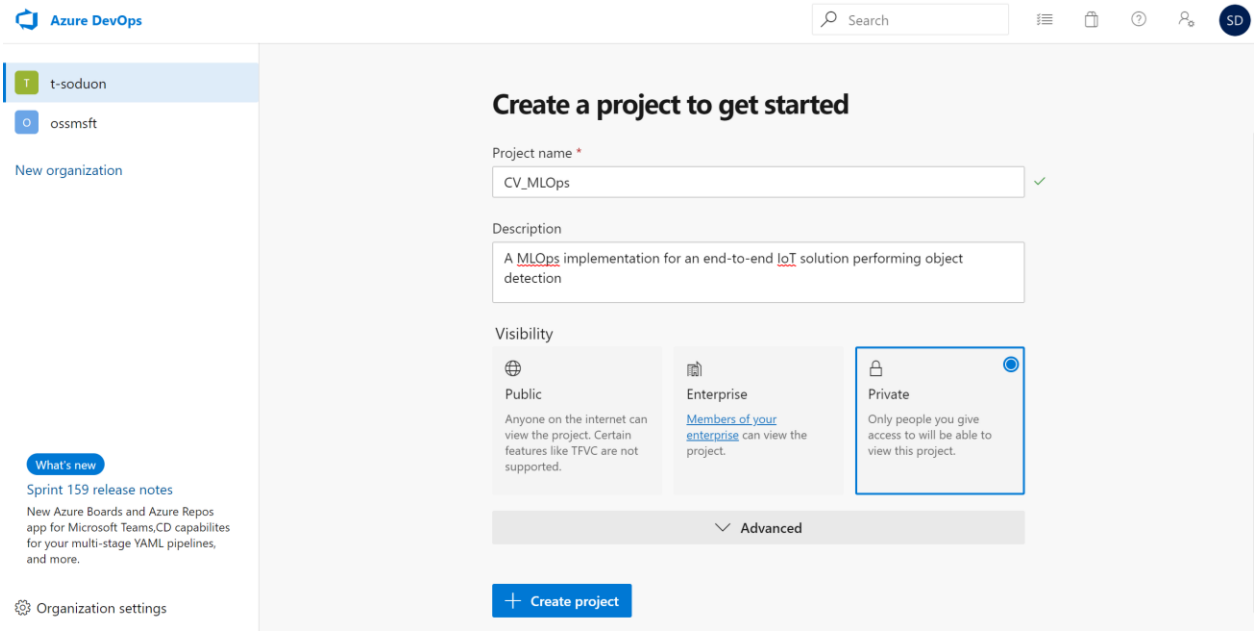
Setting up an Azure DevOps environment

To set up the DevOps environment needed for the rest of this module, perform the following steps:

1. Sign in to Azure DevOps (with your Azure account).



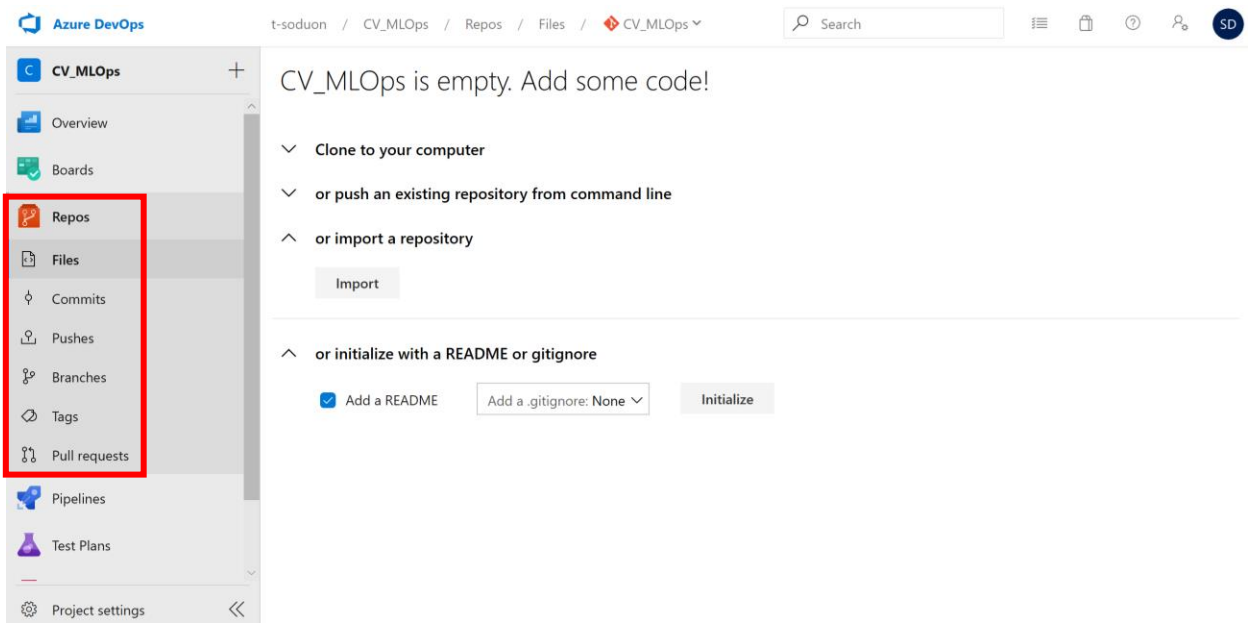
2. An organization is created based on the account you used to sign in. Sign in to your organization at any time via https://dev.azure.com/{yourorganization}. If it is the first time you sign in on your Azure DevOps account, create a new project.



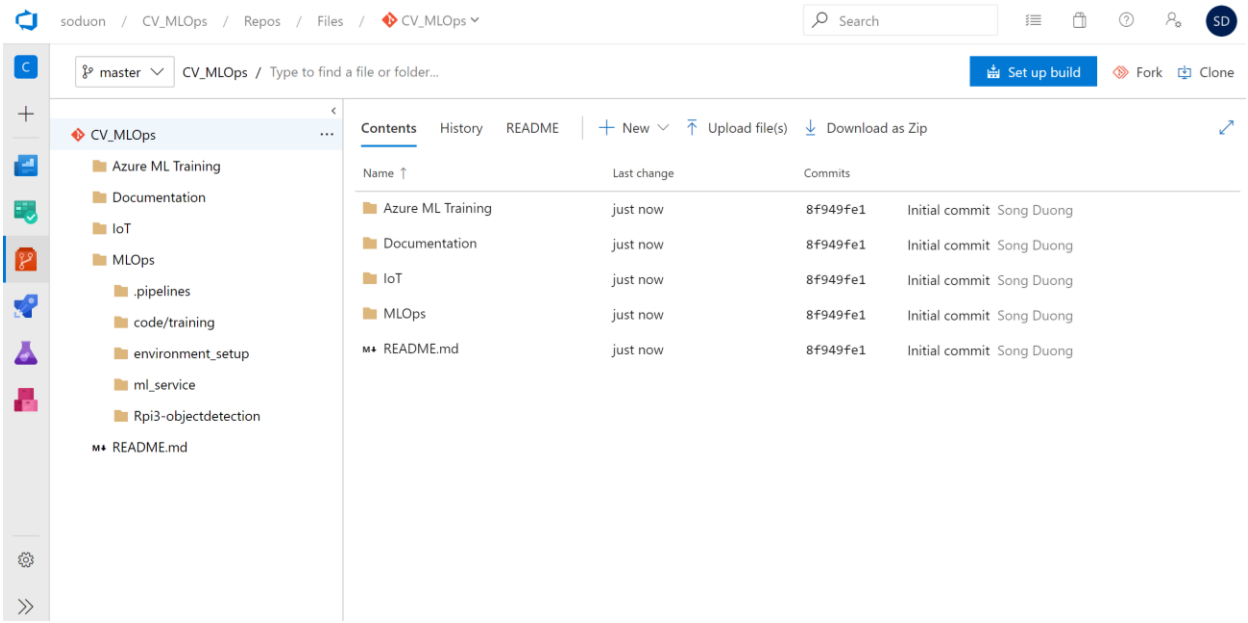
3. Once created, clone this repository:

```
git clone https://github.com/microsoft/azure-iot-edge-cv-model-samples.git
```

4. Then upload the code on your Azure repository.



5. Whether you import or push the previous repo, your source directory should look like this :



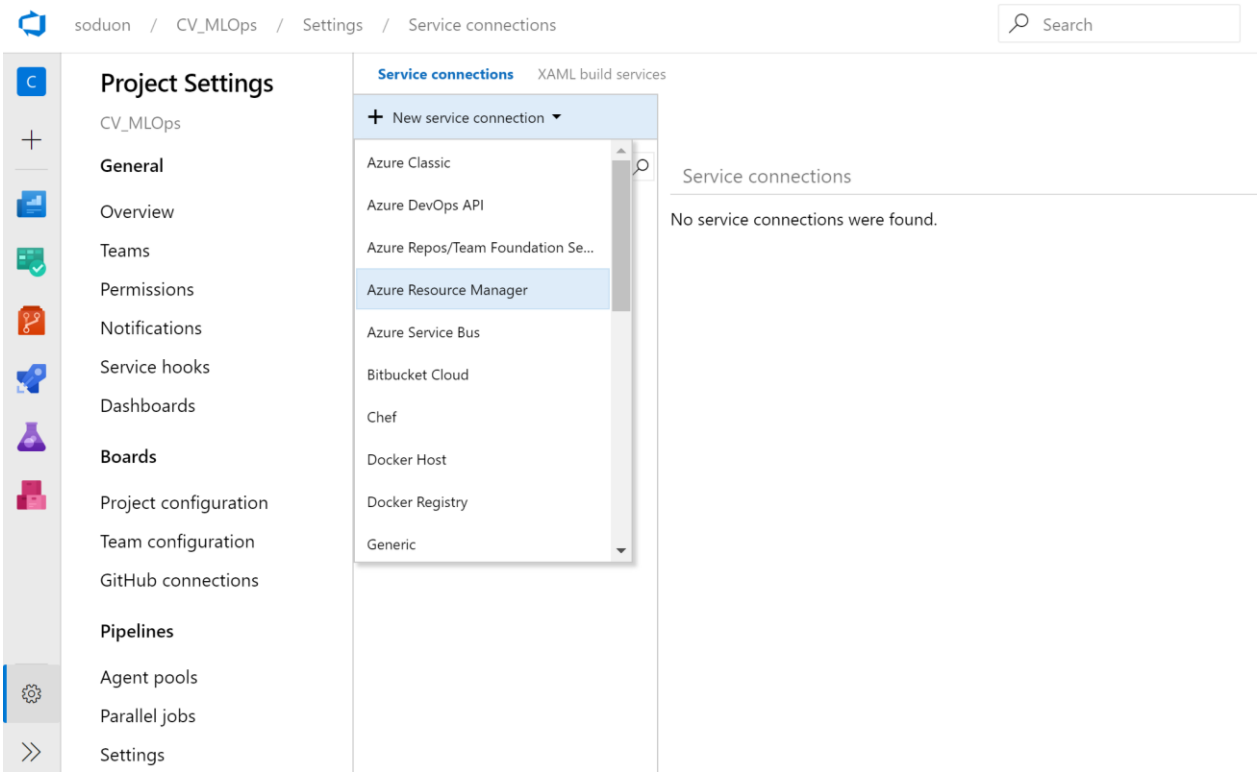
MLOps comes with the following folders :

- ***.pipelines***. Contains the pipelines needed for the steps described above.
- ***code/training***. Contains the training script and files needed for training. (This folder will typically be updated by data scientists while working on the model).
- ***environment_setup***. Contains scripts to rapidly jumpstart a fresh MLOps/IoT environment packaged in one resource group on Azure to facilitate management and deletion if needed.
- ***ml_service***. Contains utility scripts to manage training on Azure ML.

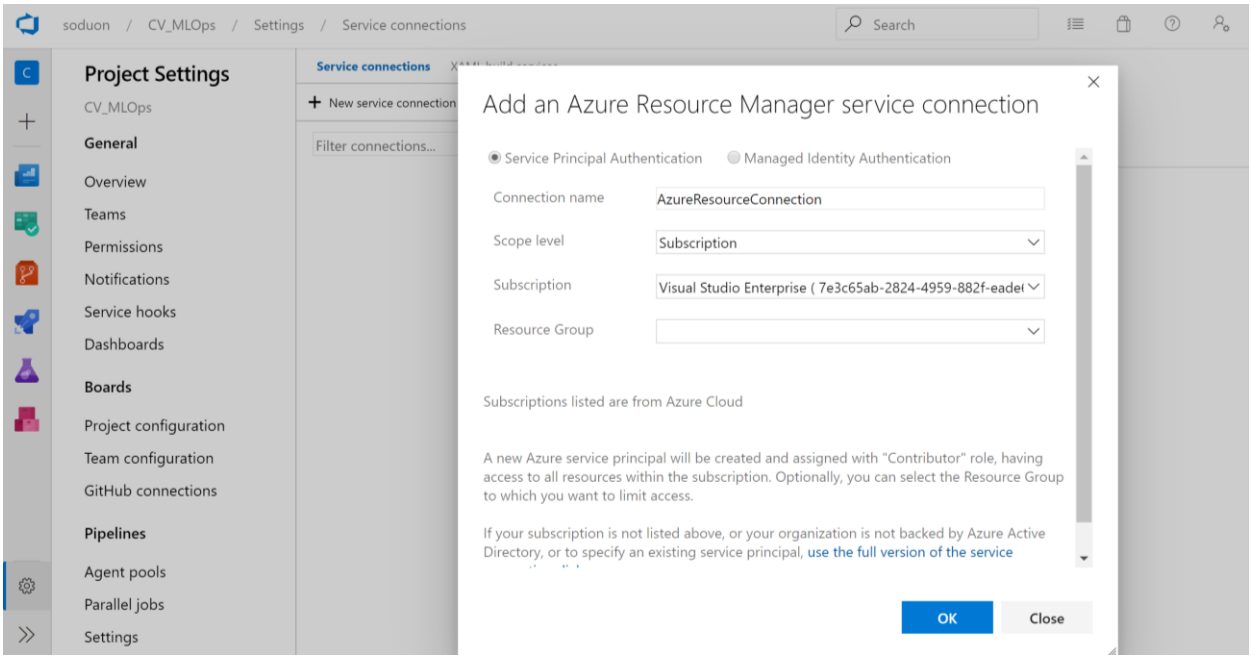
- ***Rpi3-objectdetection***. contains the IoT solution developed in the previous modules.

Before diving into pipelines, let's set up the Azure Service connection needed to allow Azure Devops to have the rights it needs to access to your Azure services.

6. Hit the wheel on the bottom left, go to the **Service connections** section and select **Azure Resource Manager** when creating **New service connection**.



7. Name it **AzureResourceConnection**, select **Subscription for the scope** and your actual subscription (don't forget to tick **Allow all pipelines to use this connection**).

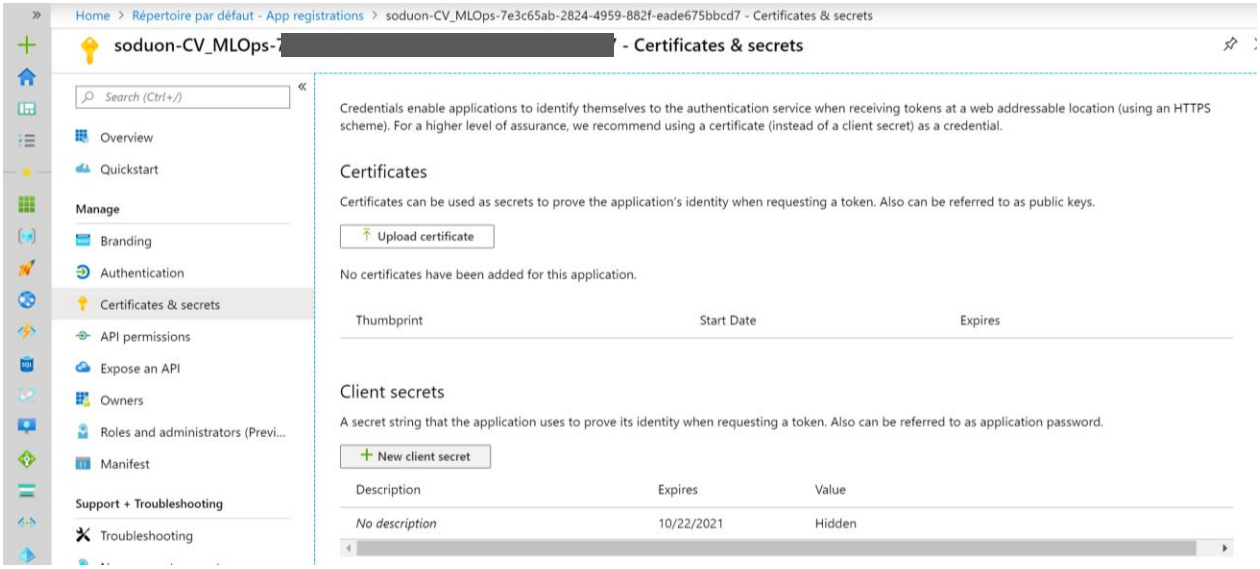


This creates a service principal on your Azure Subscription that enables connection between your Azure services. This also registers a new app inside of your Azure Active Directory (Azure AD) and grants it a Contributor access (which is fairly enough in our case).

Note You must have sufficient permissions to register an application with your Azure AD tenant, and assign the application to a role in your Azure subscription. Contact your subscription administrator if you don't have the permissions. Normally a subscription admin can create a Service principal and can provide you the details.

Now, you need to create and provide Azure DevOps an application secret

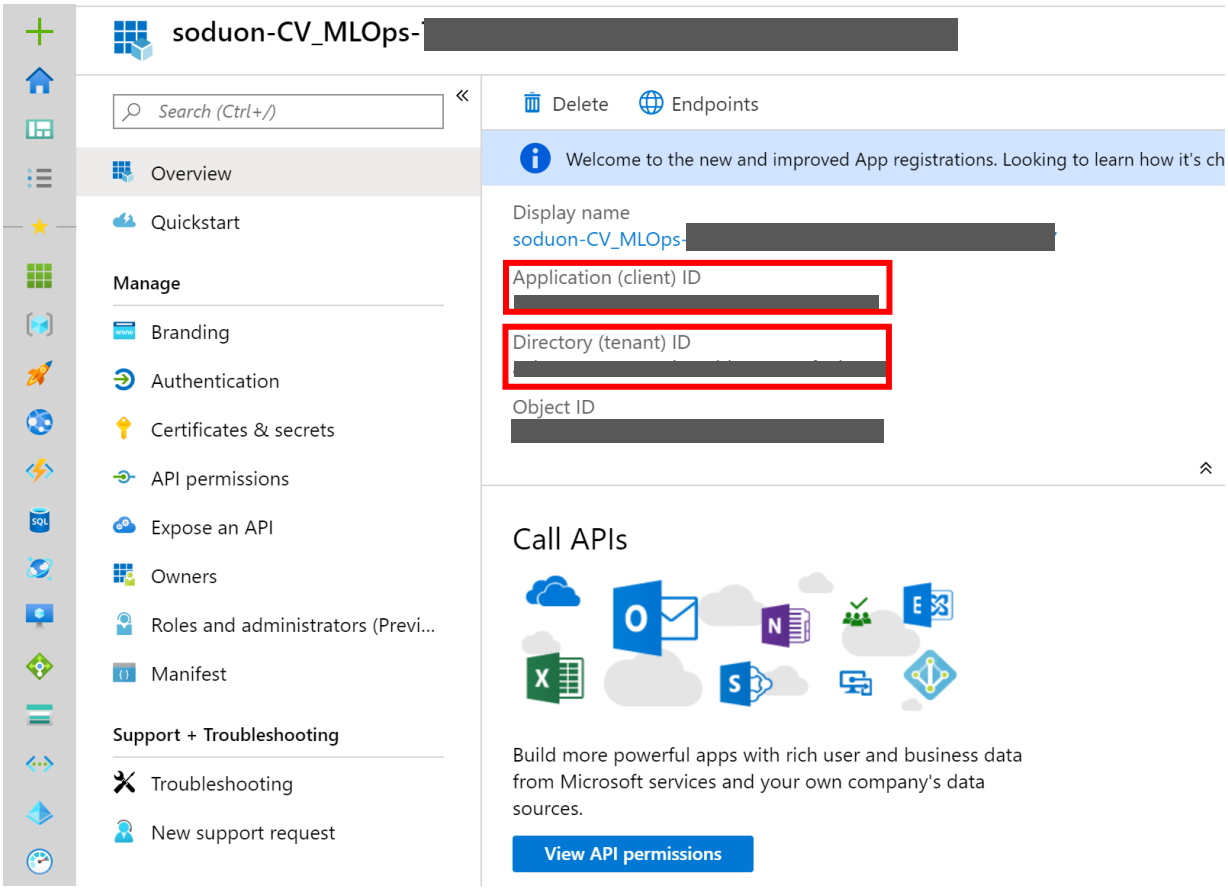
8. Sign in to the Azure portal via https://portal.azure.com and go to your Azure AD/App registrations/Certificates & secrets and select **New client secret**.



This secret string acts as a password, it is the way for Azure DevOps to access your Azure account when requesting a service. Once created, save it somewhere because it will be hidden once you refresh the page.

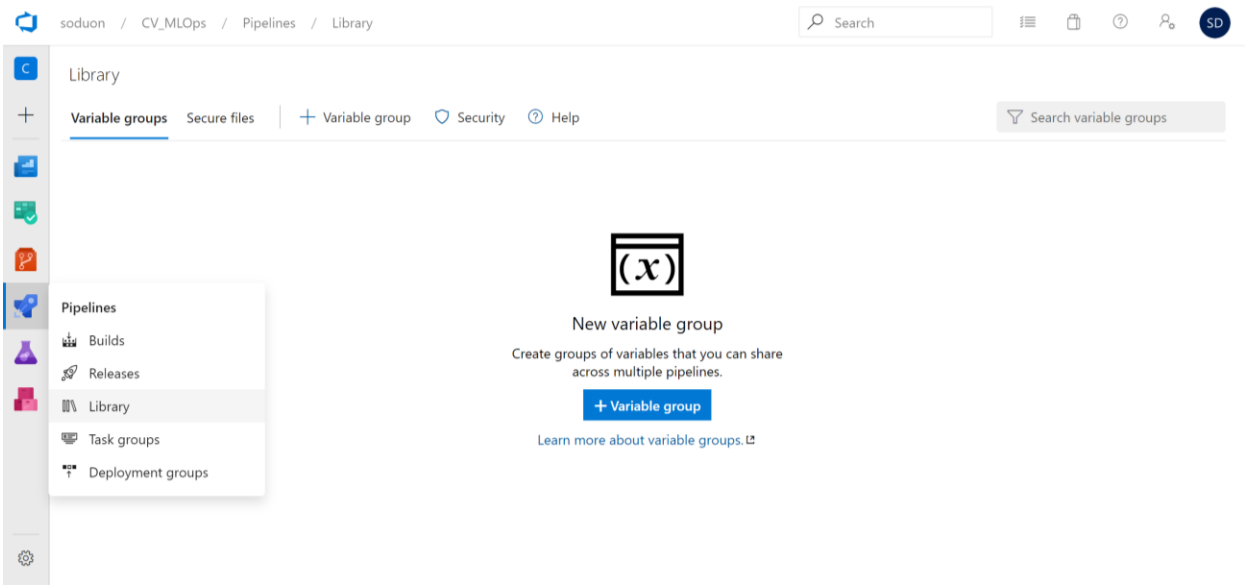
To configure the connection, you need to get :

- Application (client) ID (`SP_APP_ID` below)
- Directory (tenant) ID (`TENANT_ID` below)



Back to Azure DevOps, in order to create the necessary components, you need to set up environment variables for pipelines to work, for that, you can create variable groups to separate the different tasks.

9. On Azure DevOps, go to **Pipelines/Library** and create 2 variable groups: one for AzureML and another for Azure IoT Edge.



- `devopsforai-aml-vg`

Name	**Suggested value**
AML_COMPUTE_CLUSTER_CPU_SKU	STANDARD_NC6
AML_COMPUTE_CLUSTER_NAME	train-cluster
BASE_NAME	[unique base name like YOLOv3]
DATA_PATH_DATASTORE	VOCdevkit (**this folder needs to be uploaded before-hand by data scientists for example**)
EXPERIMENT_NAME	mlops
LOCATION	westeurope
MODEL_DATA_PATH_DATASTORE	VOCmodel_data (**this folder needs to be uploaded before-hand by data scientist for example**)
MODEL_NAME	yolo.h5
SOURCES_DIR_TRAIN	MLOps/code
SP_APP_ID	[see above]
SP_APP_SECRET	[secret string generated] (**make it secret**)
SUBSCRIPTION_ID	[your subscription id]
TENANT_ID	[see above]
TRAIN_SCRIPT_PATH	train.py
TRAINING_PIPELINE_NAME	training-pipeline

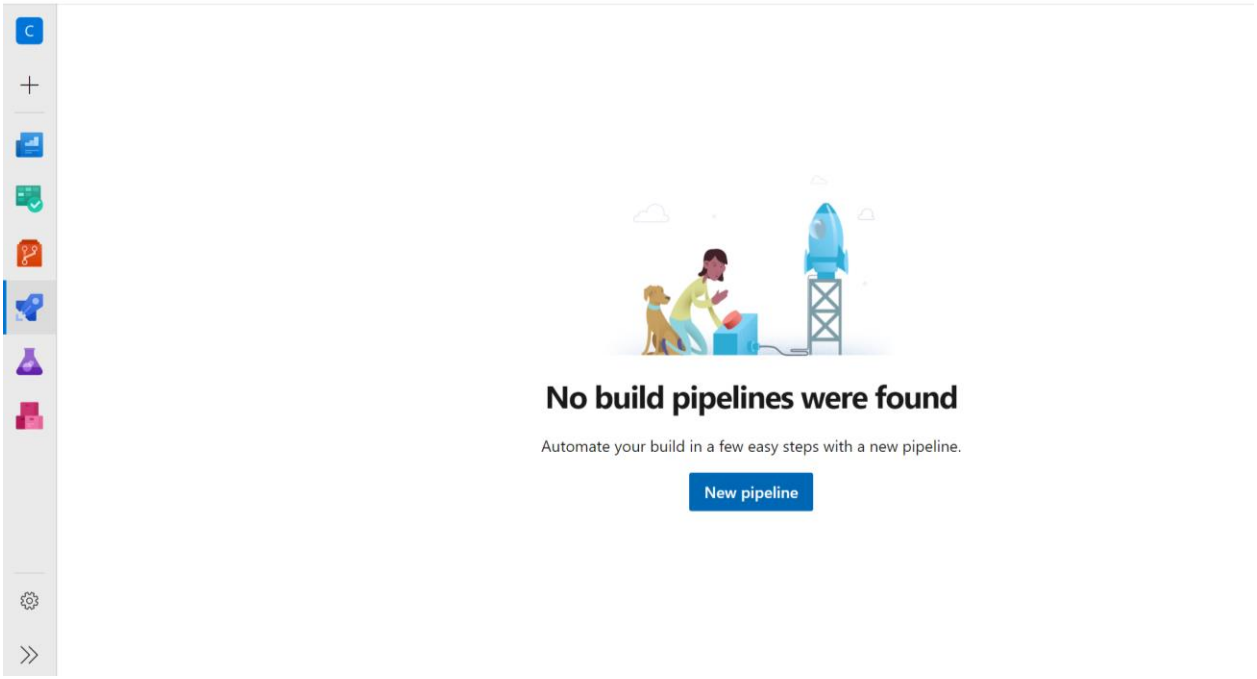
- **`iotedge-vg`**

Name	Value
ACR_ADDRESS	[to be specified later on]
ACR_PASSWORD	[to be specified later on]
ACR_USER	[to be specified later on]
EDGE_DEVICE_ID	raspicam
IOTHUB_NAME	YOLOv3-Hub

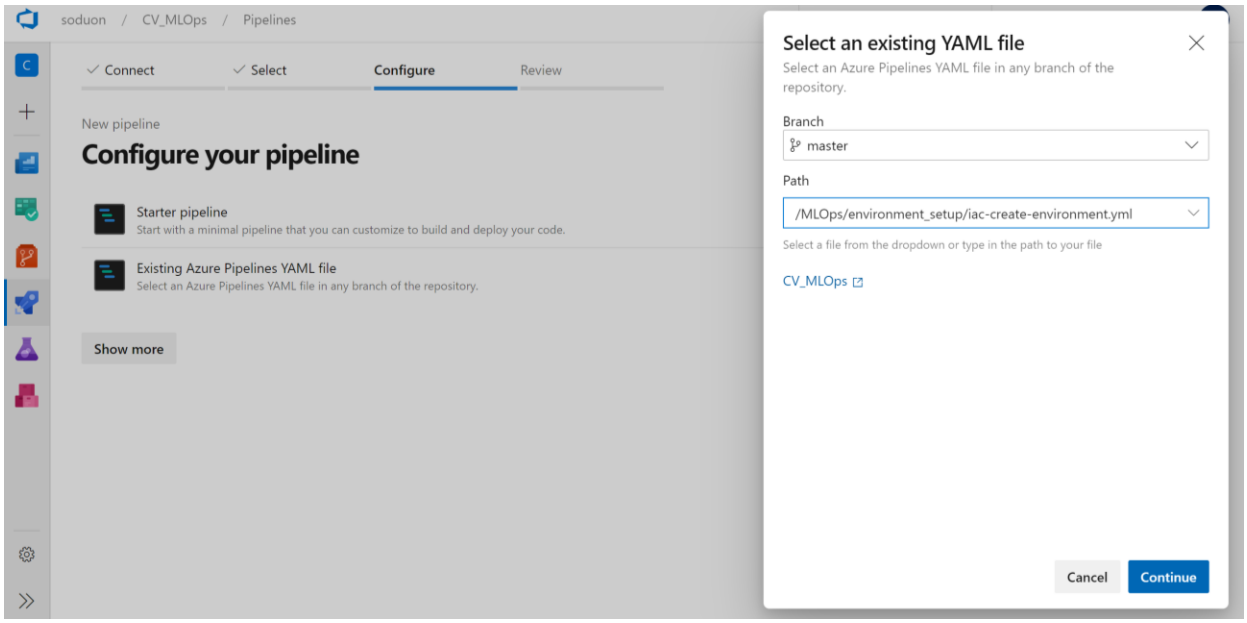
Make sure to select the **Allow access to all pipelines** checkbox in the variable group configuration.

Now, you are all set to create all the resources you need to configure your MLOps.

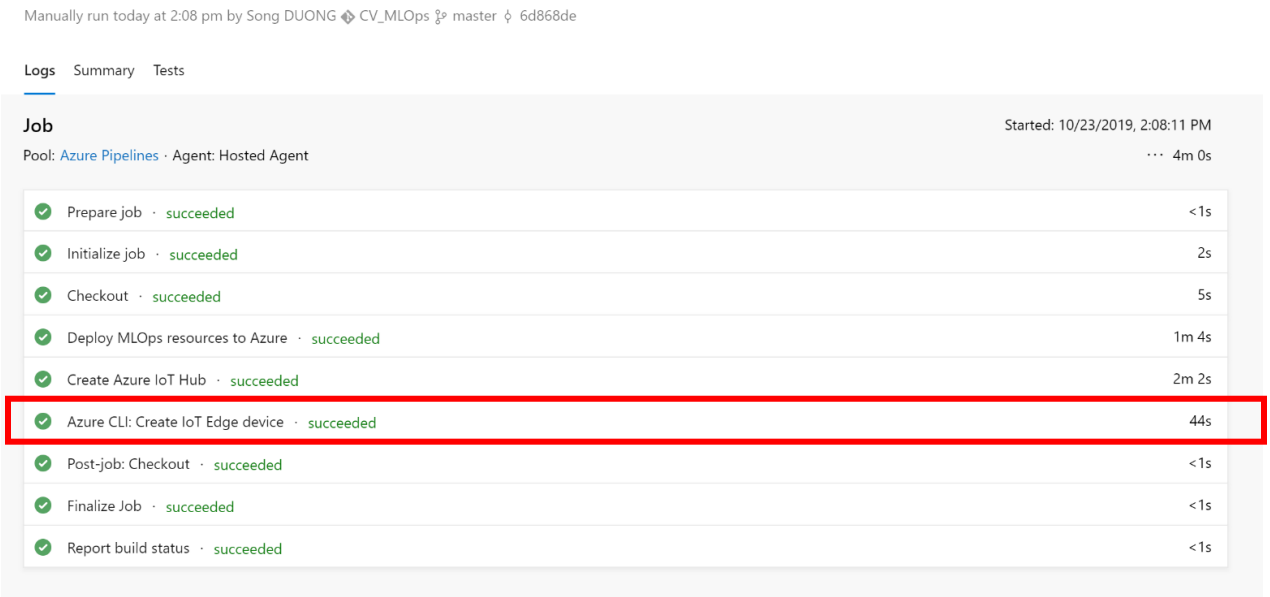
10. Go to the **Build pipelines** section and create a new pipeline.



11. Select **Azure Repos Git > Existing Azure Pipelines YAML file**.



12. Hit **Continue** and **Run** to create the resource groups.

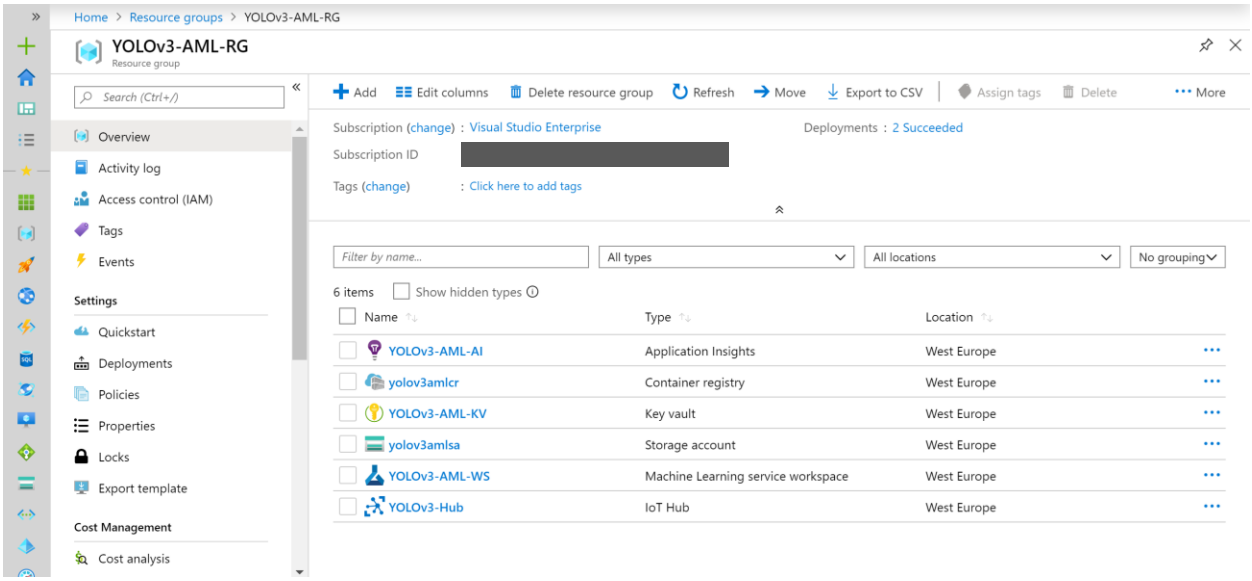


You can check on your Azure portal that a new resource group has been created containing the needed restources. In order to quickly test things out, it is a good habit as a DevOps engineer to set up "Infrastructure as Code" (IaC) pipeline that creates all required resources basing on these ARM templates[94] and this one[95].

[94] MLOpsPython: https://github.com/microsoft/MLOpsPython/blob/master/environment_setup/arm-templates/cloud-environment.json

[95] https://raw.githubusercontent.com/Azure-Samples/devops-iot-scripts/12d60bd513ead7c94aa1669e505083beaef8a480/arm-iothub.json

It also sets up a fresh IoT Hub (with "S1" as tier) and registers an IoT Edge device with "*dev*" as **tag**. You can retrieve the connection string on Azure portal or by clicking on the logs of **Azure CLI: Create IoT Edge device** (it appears at the end) and configure the connection with your IoT device (see previous modules).

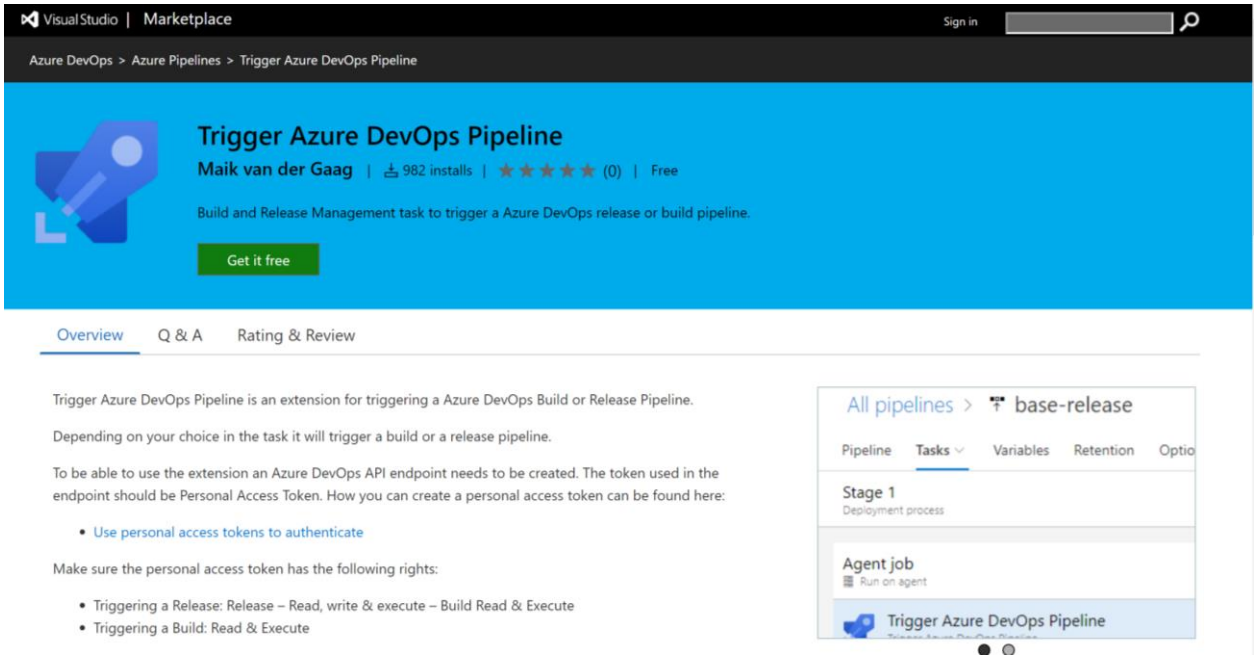


Before configuring the training pipeline, you need at some point an Azure DevOps extension that enables you to trigger a build pipeline inside of a release one.

Once you obtain the model trained and registered on AzureML, you need to trigger a build pipeline to build & push the modules packaged with the new YOLOv3 model, you will see that later on. For that, you need to follow the next steps.

13. Go on Azure Marketplace[96] and select **Get it free** and hit **Install** (*Thank you Maik van der Gaag by the way*).

[96] Trigger Azure DevOps Pipeline: https://marketplace.visualstudio.com/items?itemName=maikvandergaag.maikvandergaag-trigger-pipeline

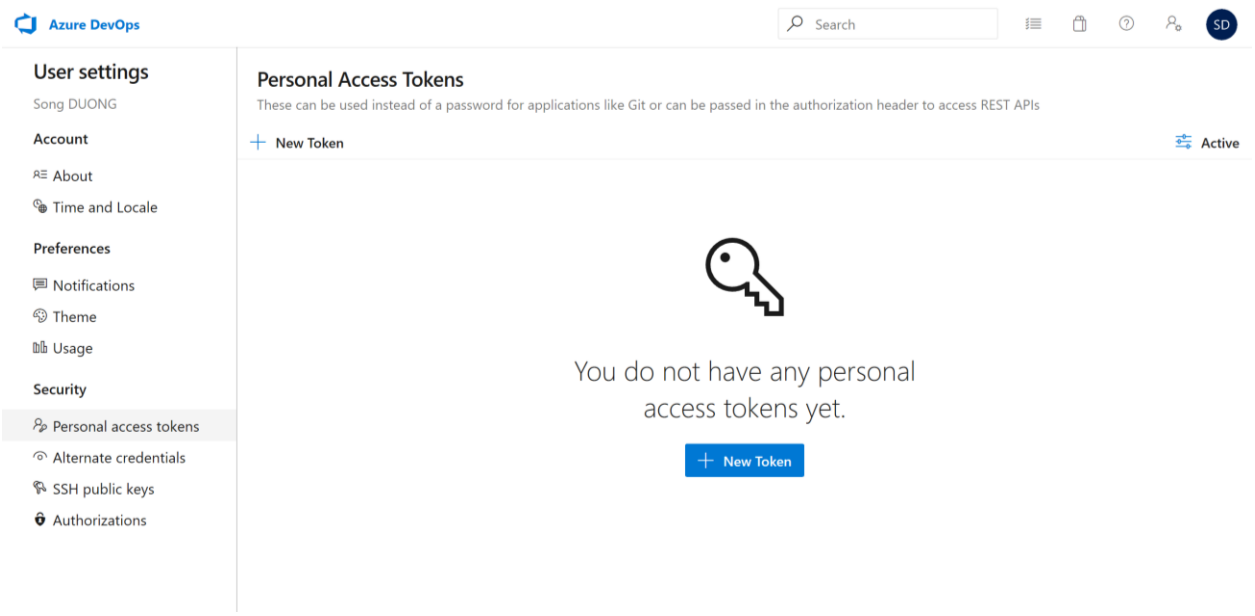


To make it work autonomously, you need to give it a Personal Access Token[97] (PAT). They are alternate passwords that you can use to authenticate into Azure DevOps (without authenticating yourself each time you need the extension).

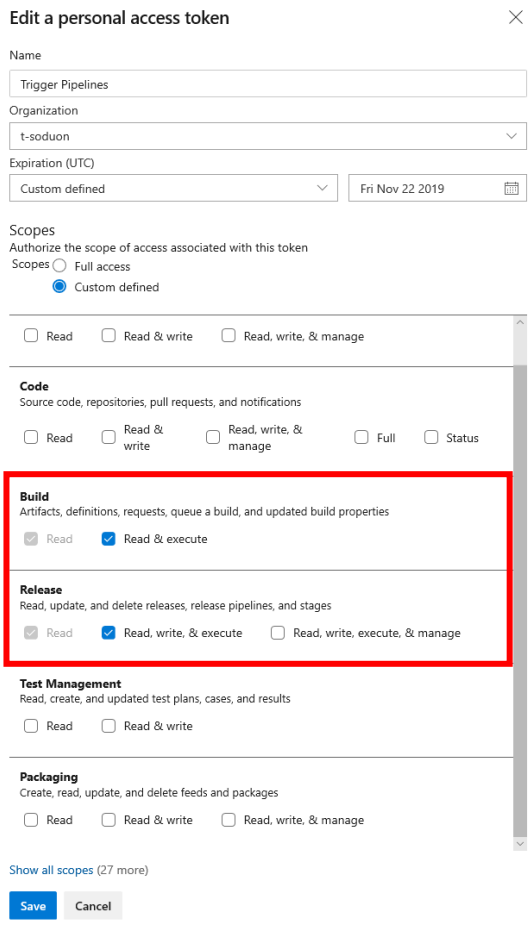
To create a PAT, from your home page:

a. Click on your profile in the top-right corner, select **Azure DevOps profile** and go to the **PAT** section.

[97] AUTHENTICATE ACCESS WITH PERSONAL ACCESS TOKENS: https://docs.microsoft.com/en-us/azure/devops/organizations/accounts/use-personal-access-tokens-to-authenticate?view=azure-devops&viewFallbackFrom=vsts

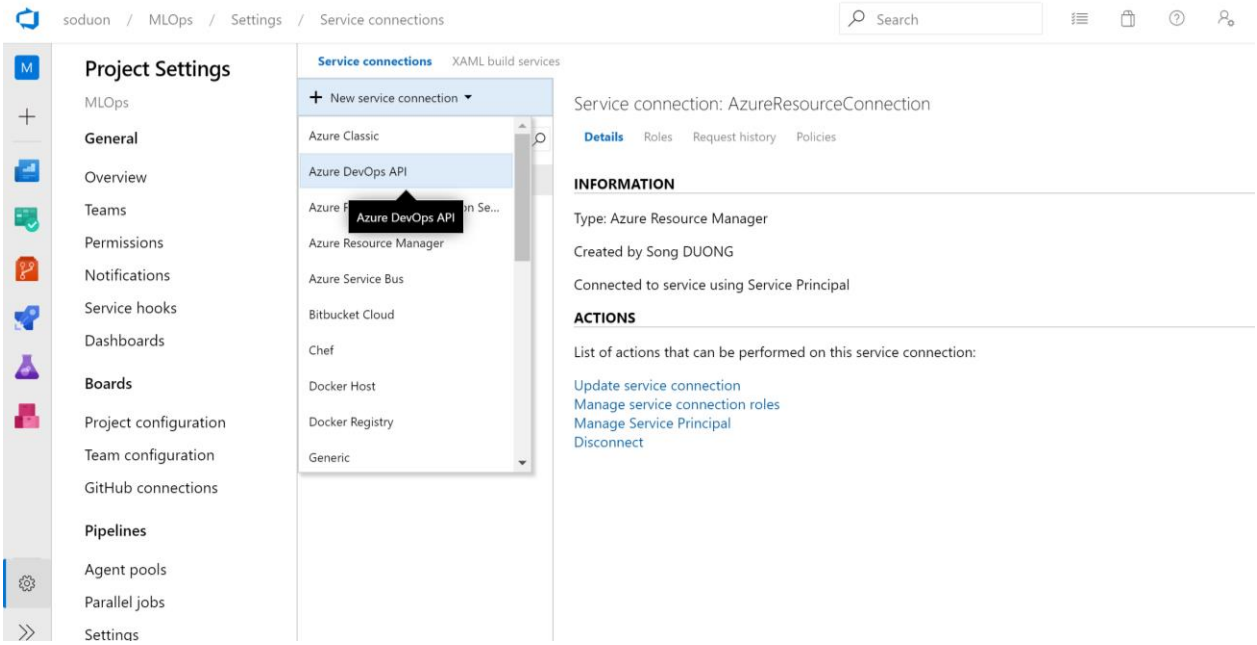


b. Select **New Token** and give it the following permissions and hit **Save**.

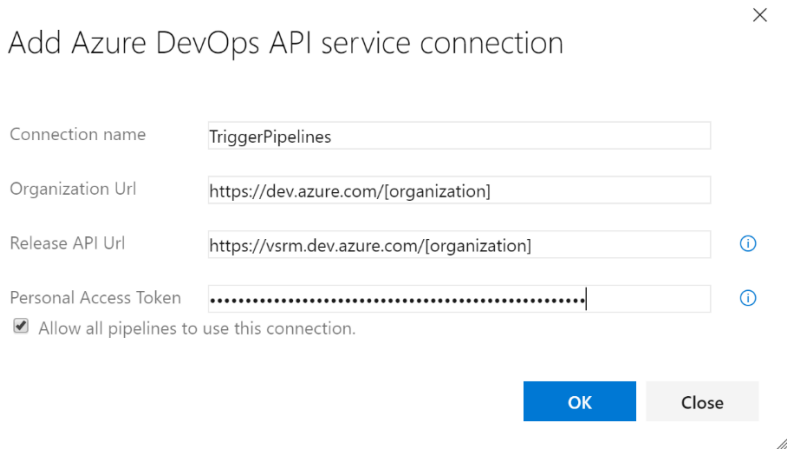


c. Save the token, you will need it to configure the service connection for the extension to work.

14. Create a new service connection as for your Azure resources but this time select **Azure DevOps API**.



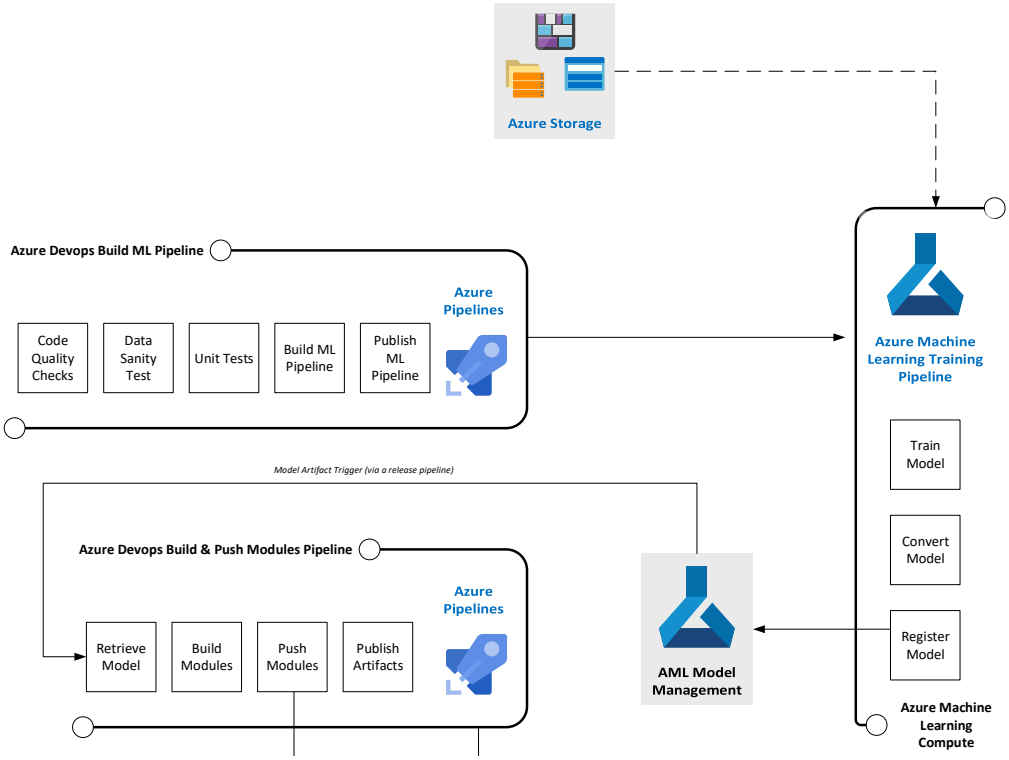
15. Fill in the values where [organization] is the name of your organization and specify your PAT in its respective box. Don't forget to to select the **Allow access to all pipelines** checkbox.



That's it, now, you are all set to configure your pipelines.

Configuring the training pipeline

The training pipeline is the CI trigger for this entire MLOps implementation. Once data scientists upload data and commit changes to the repository (whether it is to run tests or rewrite the training script), a first Build pipeline is triggered: it creates and publishes a ML pipeline on Azure ML as well as a build artifact to trigger a second (Release this time!) pipeline invoking the training.



In this guide, you are training with this Keras implementation of YOLOv3[98]. Typically, if data scientists wish to retrain the model on another dataset (VOC dataset for instance), they need to provide a folder (here it is VOCmodel_data) which contains:

VOCmodel_data

```
└── classes.txt
└── yolo_weights.h5
└── yolo_anchors.txt
```

Where *classes.txt* contains the labels of the detected objects, *yolo_weights.h5* contains the weights of the original YOLOv3 pre-trained on COCO dataset and *yolo_anchors.txt* contains the anchors for YOLO to detect potential boxes on different scale.

The training script also needs an annotation text file that contains the path to each image, the ground-truth boxes as well as the id of the corresponding objects.

```
1. path/to/img1.jpg 50,100,150,200,0 30,50,200,120,3
2. path/to/img2.jpg 120,300,250,600,2
3. ...
```

See here[99] for more info about how to train.

---

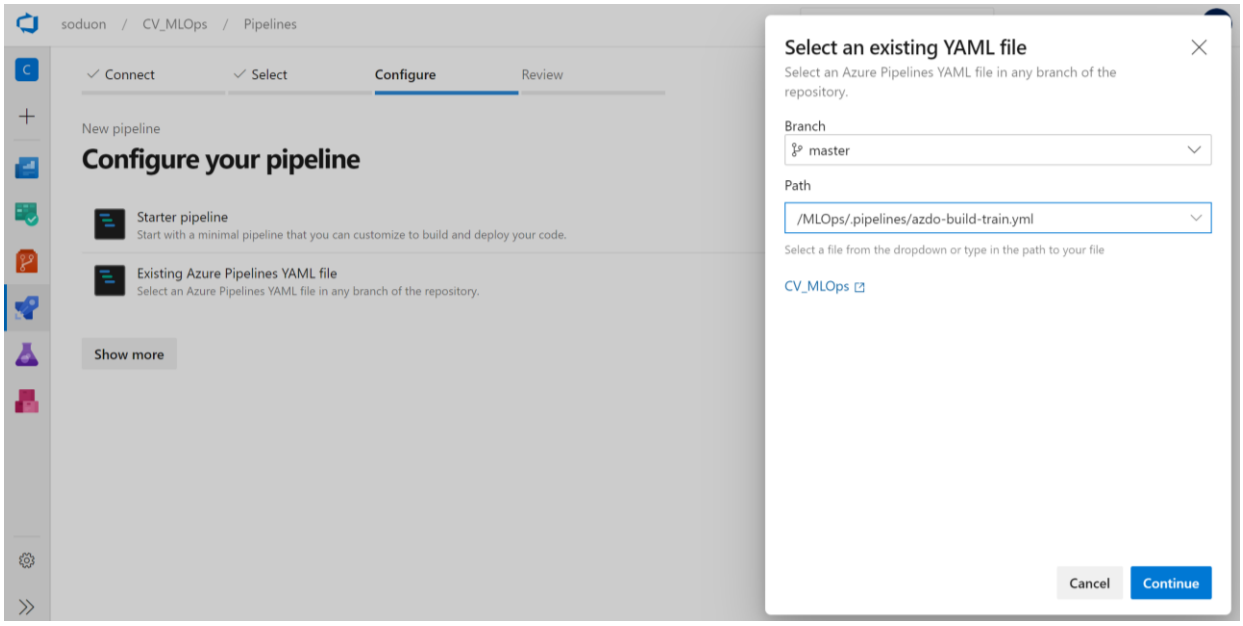
[98] A Keras implementation of YOLOv3 (Tensorflow backend): https://github.com/qqwweee/keras-yolo3

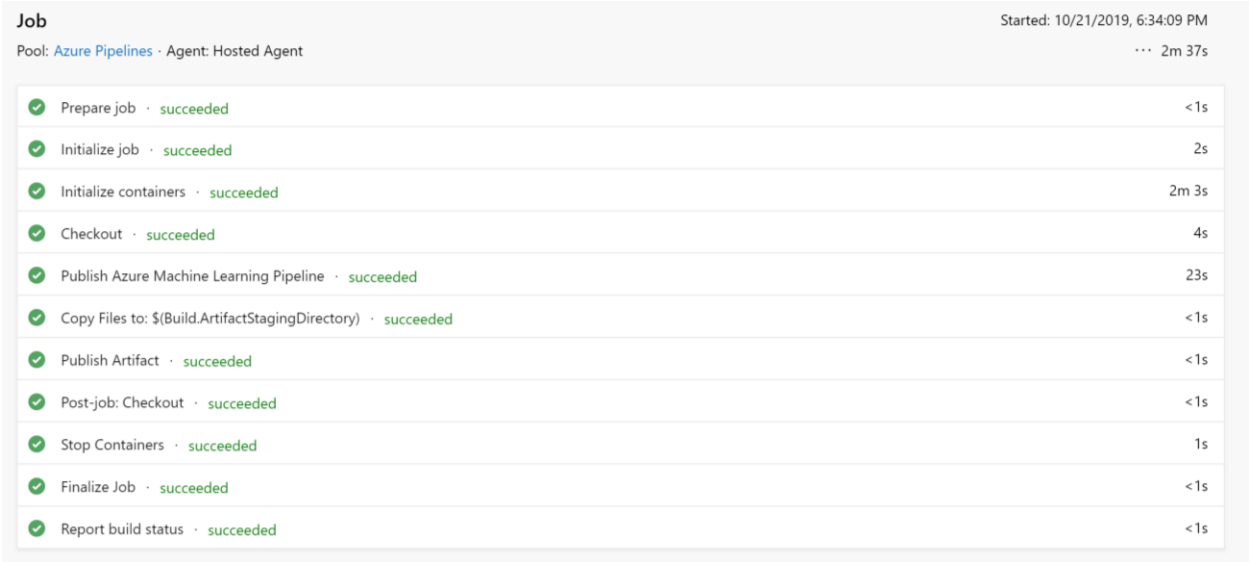
[99] https://github.com/qqwweee/keras-yolo3/blob/master/README.md

Note You need to upload to your datastore these two folders here to make it work. You can download them here[100] and here[101], extract the folders and upload them as in Module 0: Preparing your Computer Vision model (optional), see section § Training the model with Azure ML.

To configure this pipeline, perform the following steps:

1. Create a new Build pipeline and select *azdo-build-train.yml* instead.

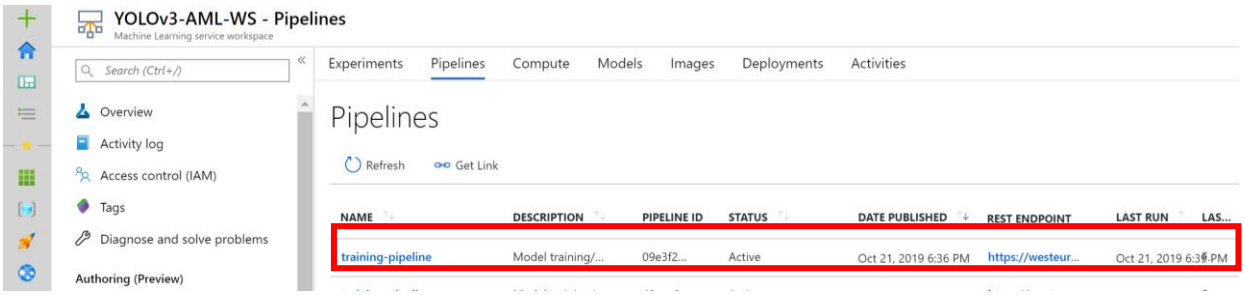




[100] https://azurecviotedge.blob.core.windows.net/mlops/VOCdevkit.zip

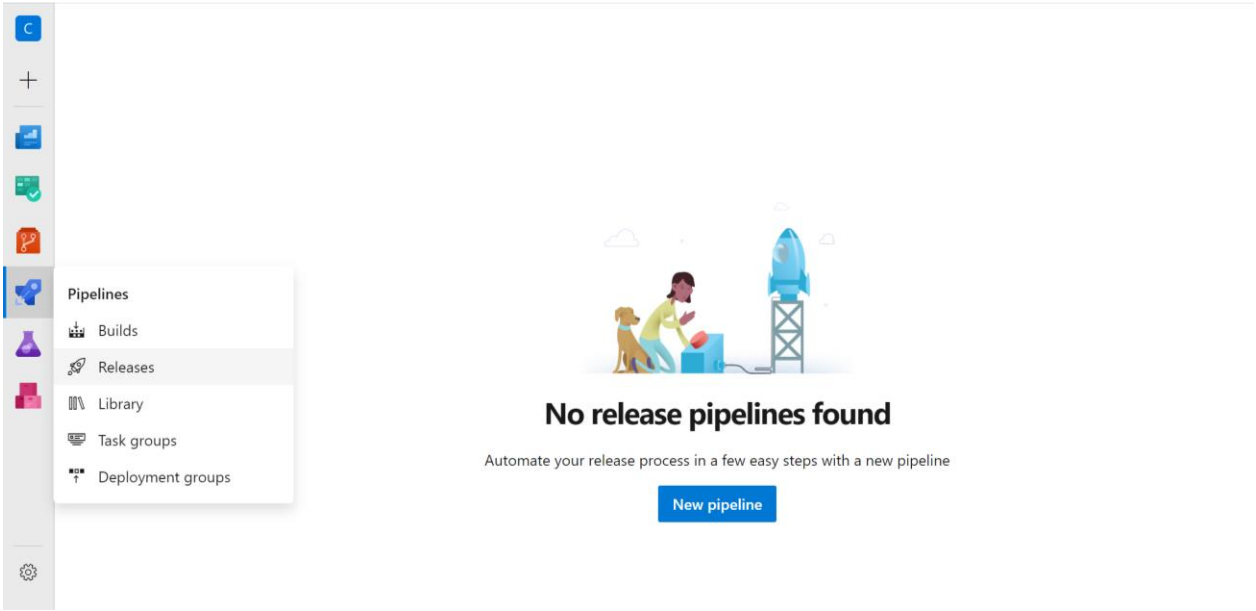
[101] https://azurecviotedge.blob.core.windows.net/mlops/VOCmodel_data.zip

You should then see on your Azure portal that another ML pipeline '`training-pipeline`' has been created.

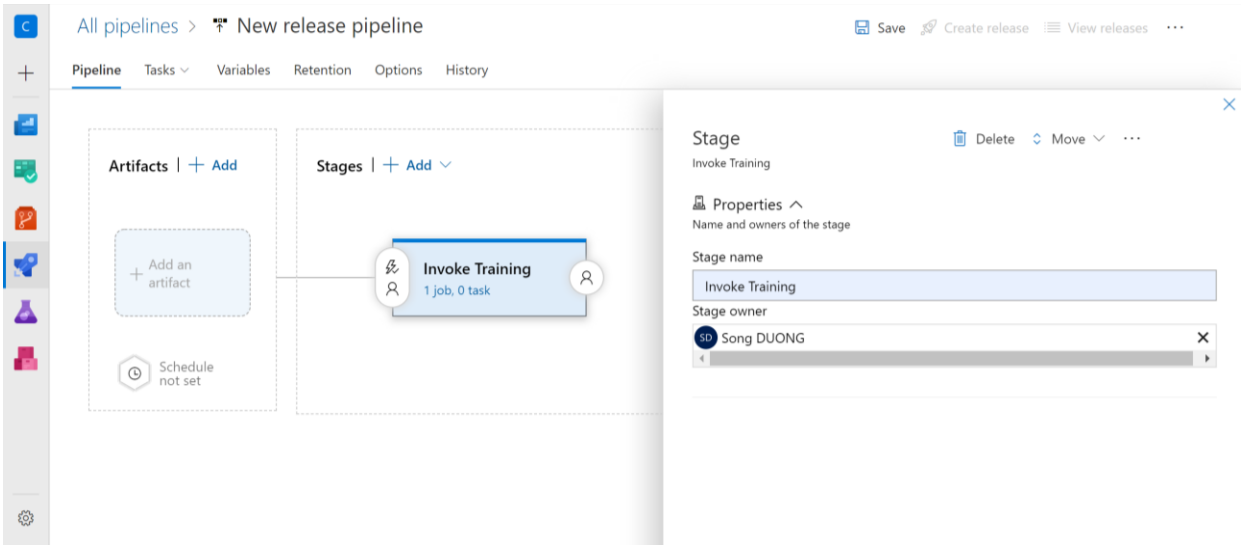


Now, let's set up a Release pipeline that will invoke the training.

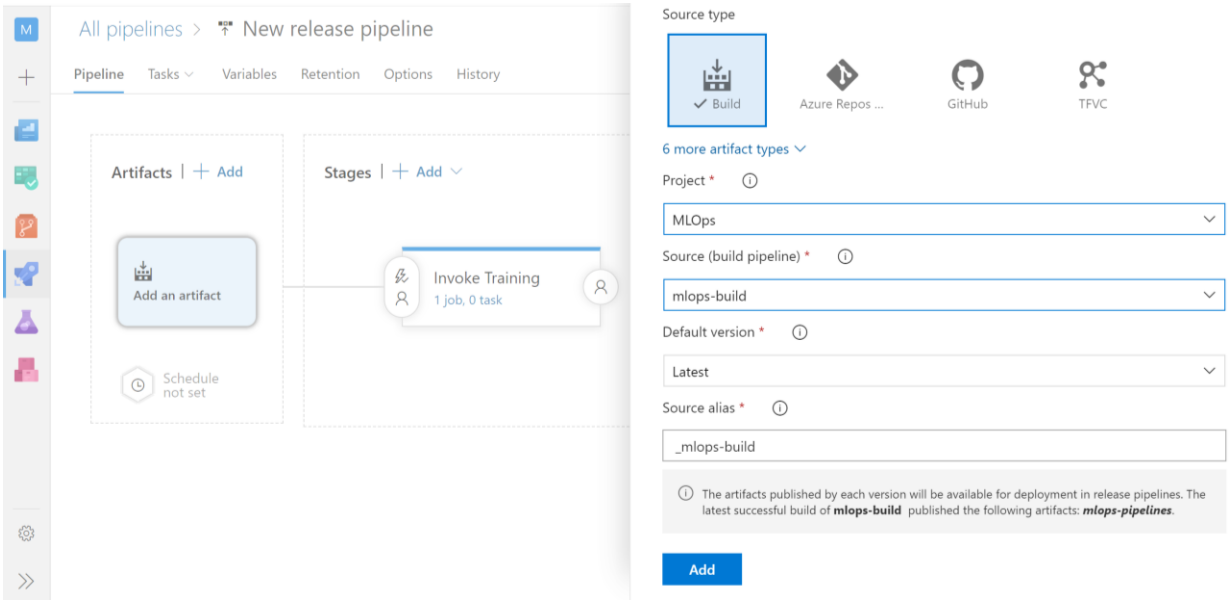
2. For that, go to **Releases** and create a new Release pipeline.



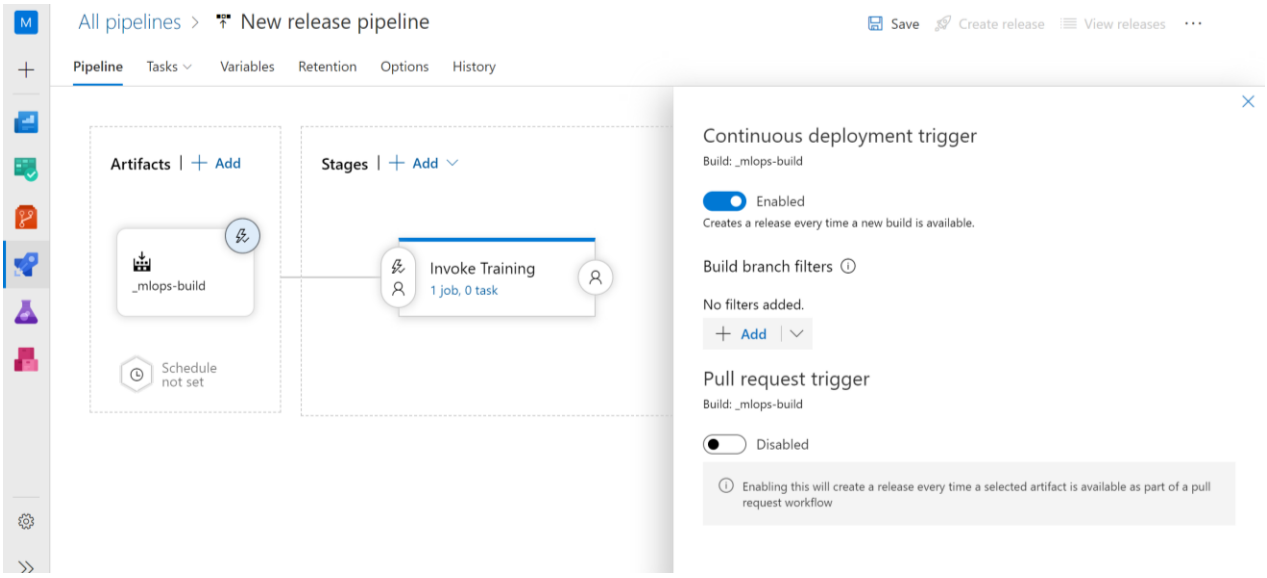
3. Start with an empty job and rename the stage "*Invoke Training*" for clarity.



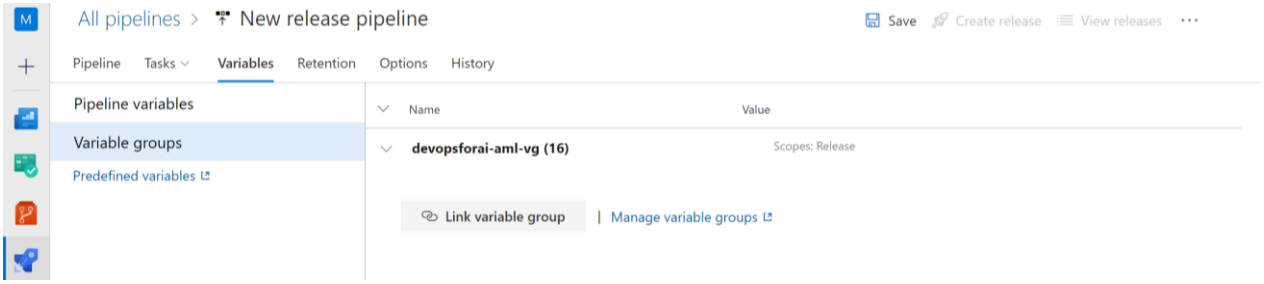
4. Select **Add an artifact** and select your build artifact (replace *mlops-build* with your build artifact) and hit Add.



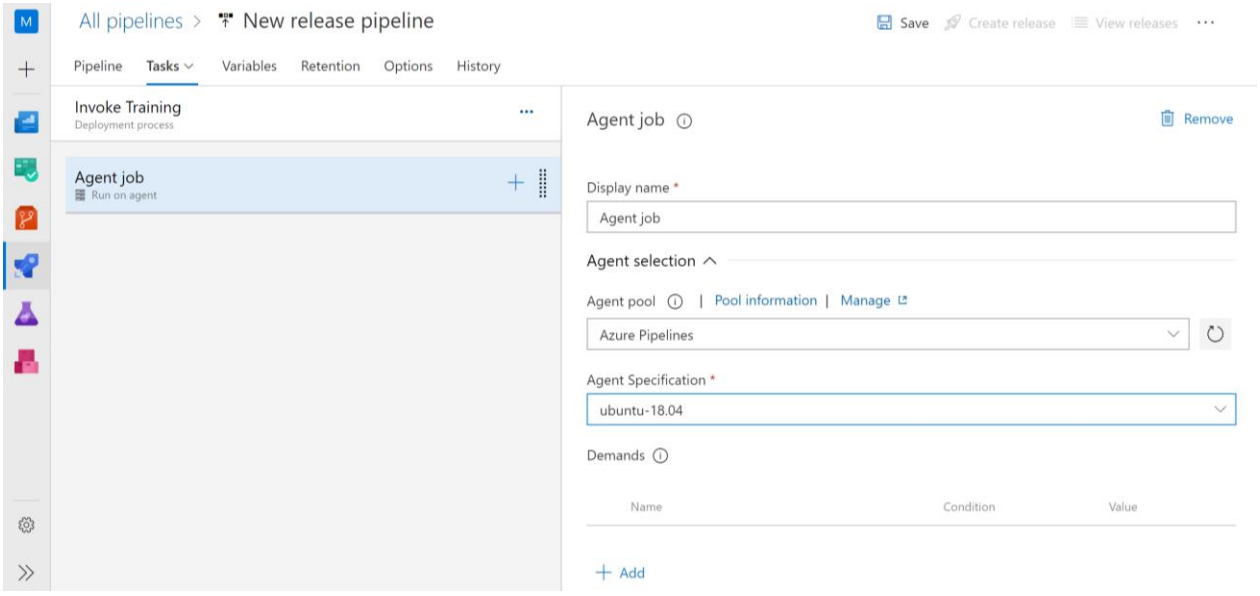
5. Click on the lighting bolt icon next to the artifact to enable Continuous Deployment.



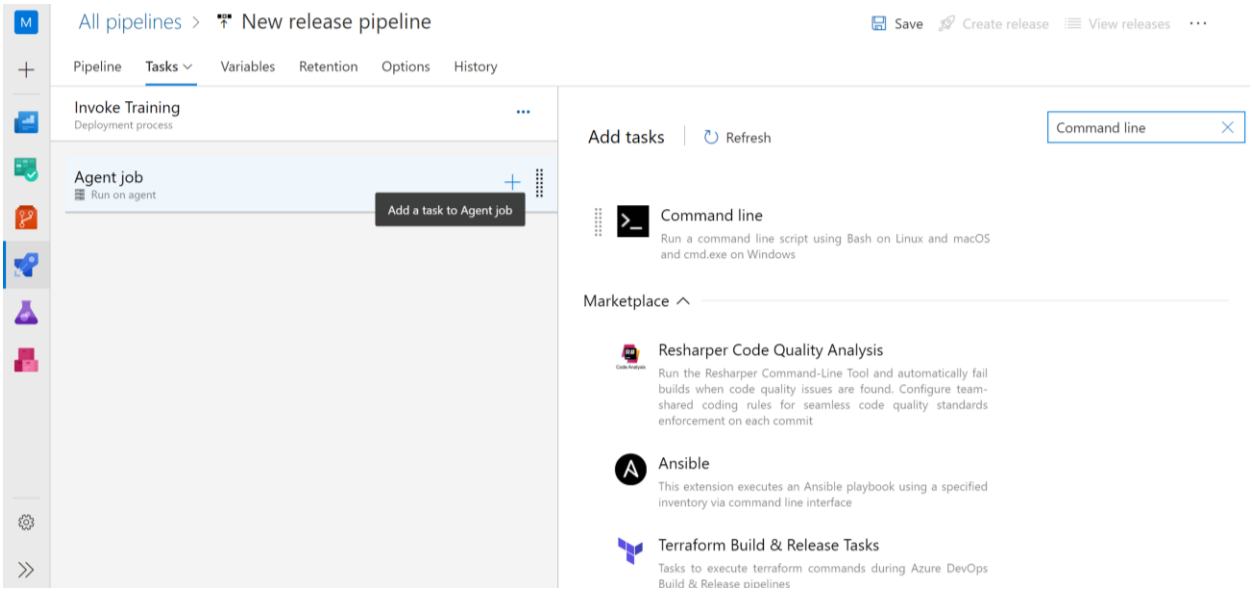
6. Click on the **Variables > Variable groups**, and to the right, click on **Link variable group**. From there, pick the `devopsforai-aml-vg` variable group you created earlier, choose **Release** as a variable group scope, and click on **Link**:



7. Go to **Tasks** and add an Ubuntu 18.04 agent.



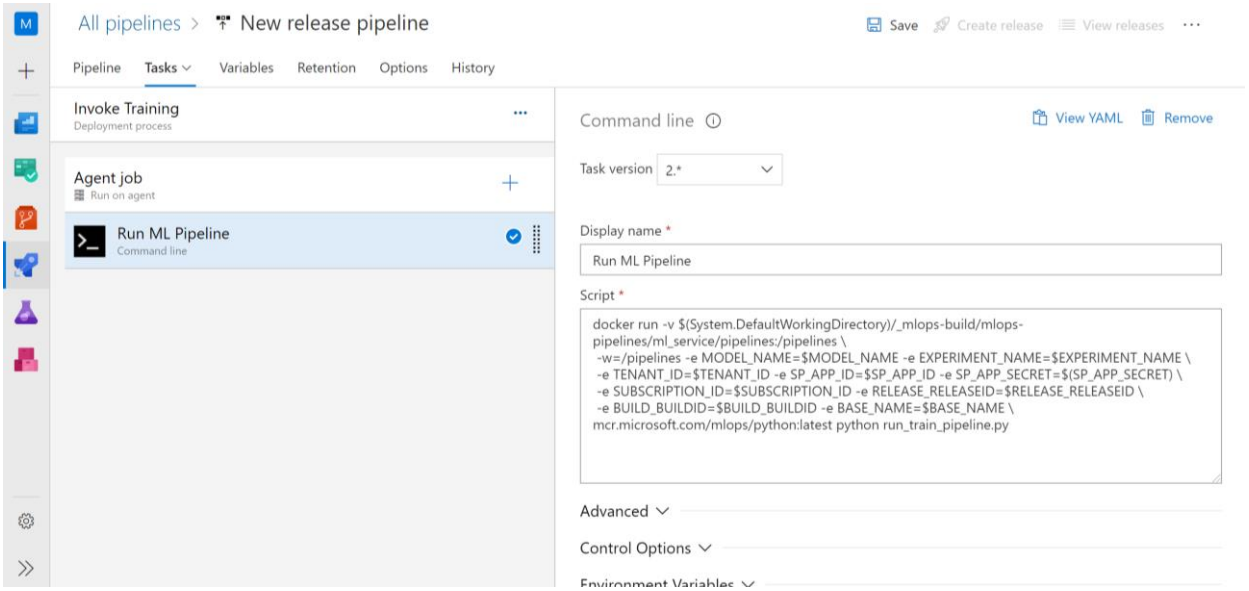
8. Add a **Command line** task.



9. Copy-paste the following command line. Basically, it passes the necessary env variables for the Docker image to run a Python script invoking the ML pipeline on Azure ML.

```
docker run -v $(System.DefaultWorkingDirectory)/_mlops-build/mlops-
pipelines/ml_service/pipelines:/pipelines \
 -w=/pipelines -e MODEL_NAME=$MODEL_NAME -e EXPERIMENT_NAME=$EXPERIMENT_NAME \
 -e TENANT_ID=$TENANT_ID -e SP_APP_ID=$SP_APP_ID -e SP_APP_SECRET=$(SP_APP_SECRET) \
 -e SUBSCRIPTION_ID=$SUBSCRIPTION_ID -e RELEASE_RELEASEID=$RELEASE_RELEASEID \
 -e BUILD_BUILDID=$BUILD_BUILDID -e BASE_NAME=$BASE_NAME \
mcr.microsoft.com/mlops/python:latest python run_train_pipeline.py
```

(replace the `_mlops-build` with your build artifact name)



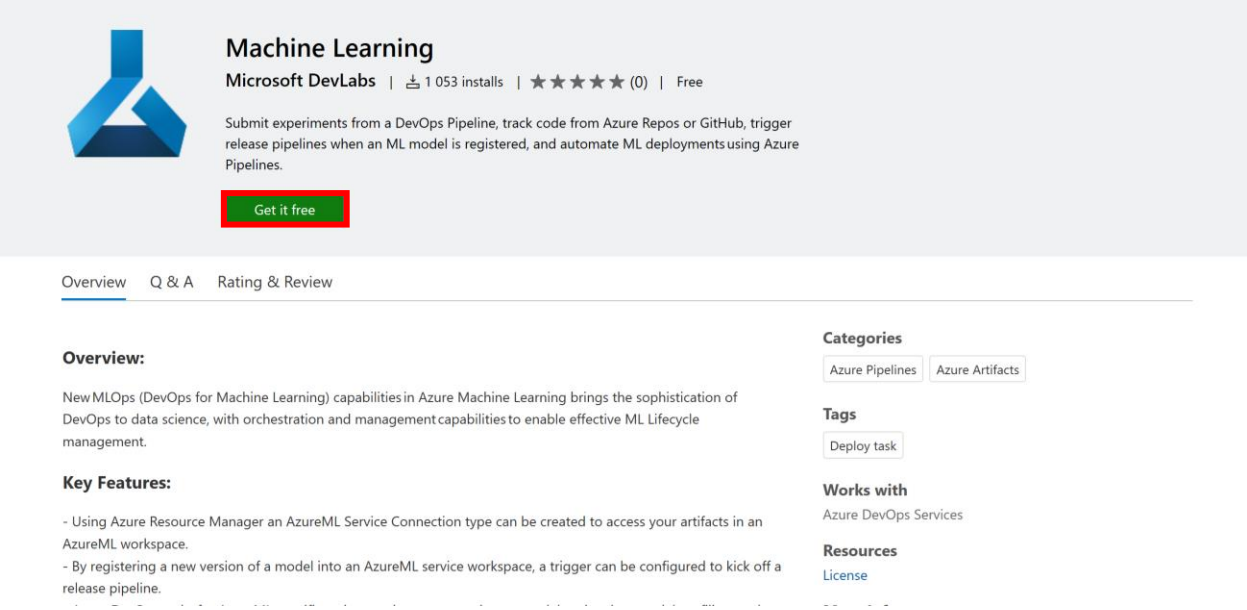
Now, choose **Create release** to start training, you can check on Azure portal that the pipeline has indeed started (it may take some time depending on how many epochs you train etc.)

Configuring the build-push-modules pipeline

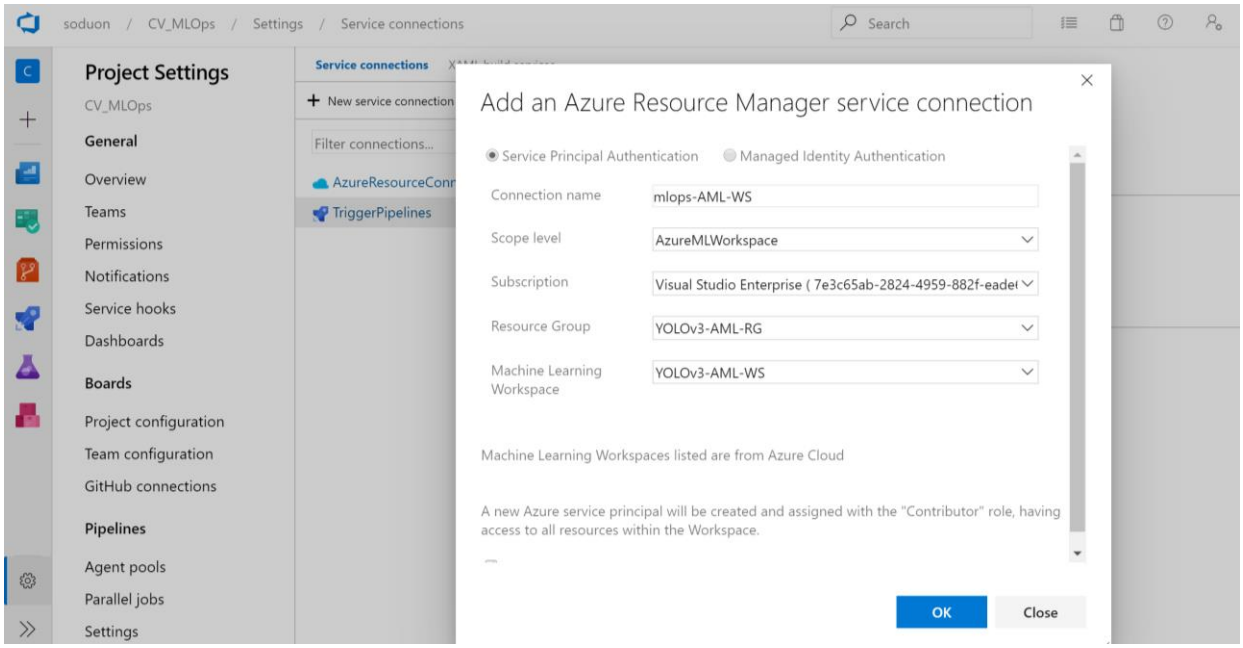
Now, you need to configure a pipeline that will trigger the build-push-modules pipeline once a new model is registered on Azure ML. For that, you can set up a release pipeline with a model artifact that will automatically trigger the `build-push-modules` pipeline every time there is a new model.

To do that , perform the following steps:

1. Install the **Azure Machine Learning** extension to your organization from the marketplace, so that you can set up a service connection to your AML workspace.

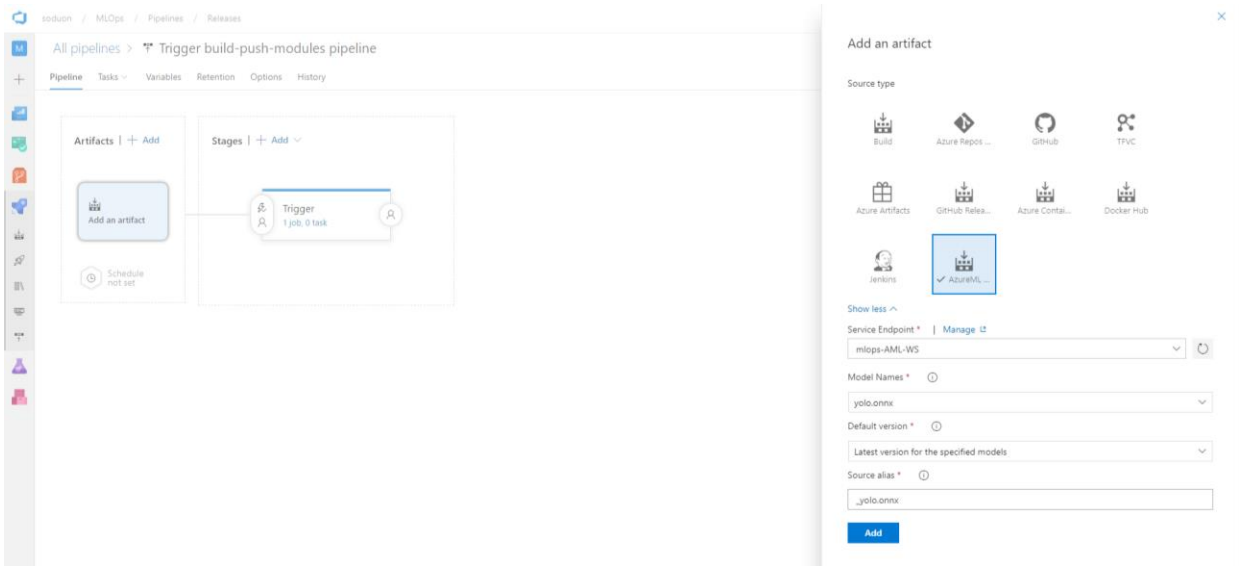


2. To configure a model artifact, there should be a service connection to your workspace. To get there, go to the project settings (by clicking on the cog wheel to the bottom left of the screen), go to the **Service Connection** section and create a **new Azure Resource Manager** as you did earlier for your Azure services but this time, choose **AzureMLWorkspace** for the scope.

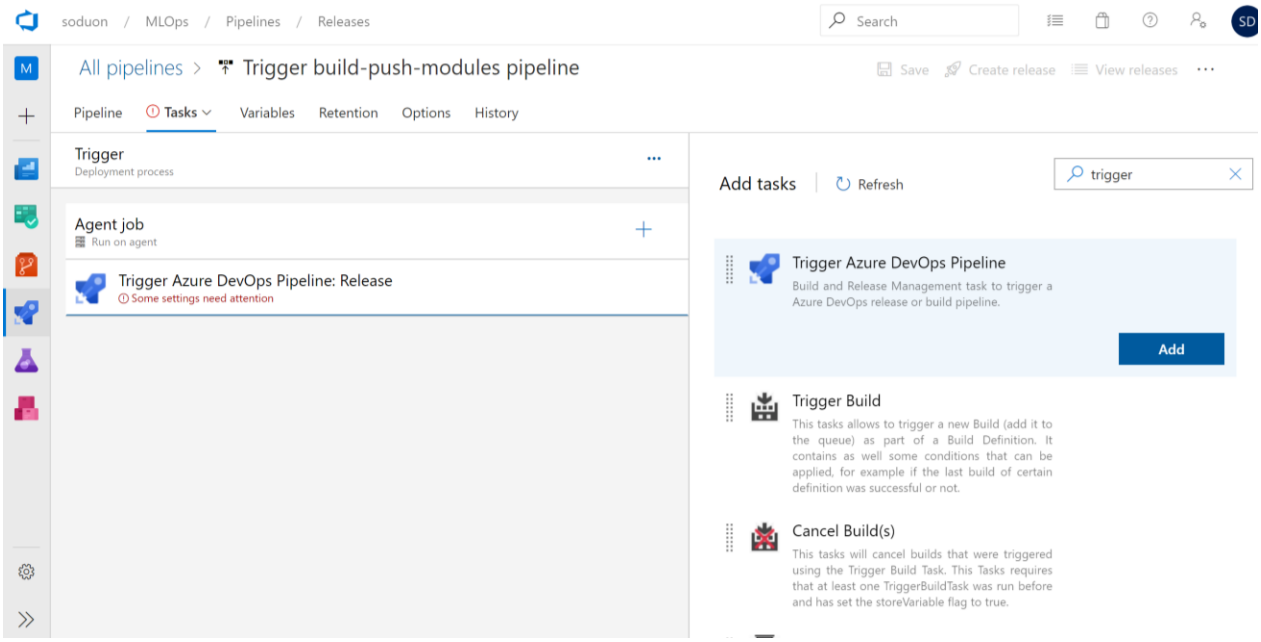


Note Creating service connection using Azure Machine Learning extension requires 'Owner' or 'User Access Administrator' permissions on the Workspace.

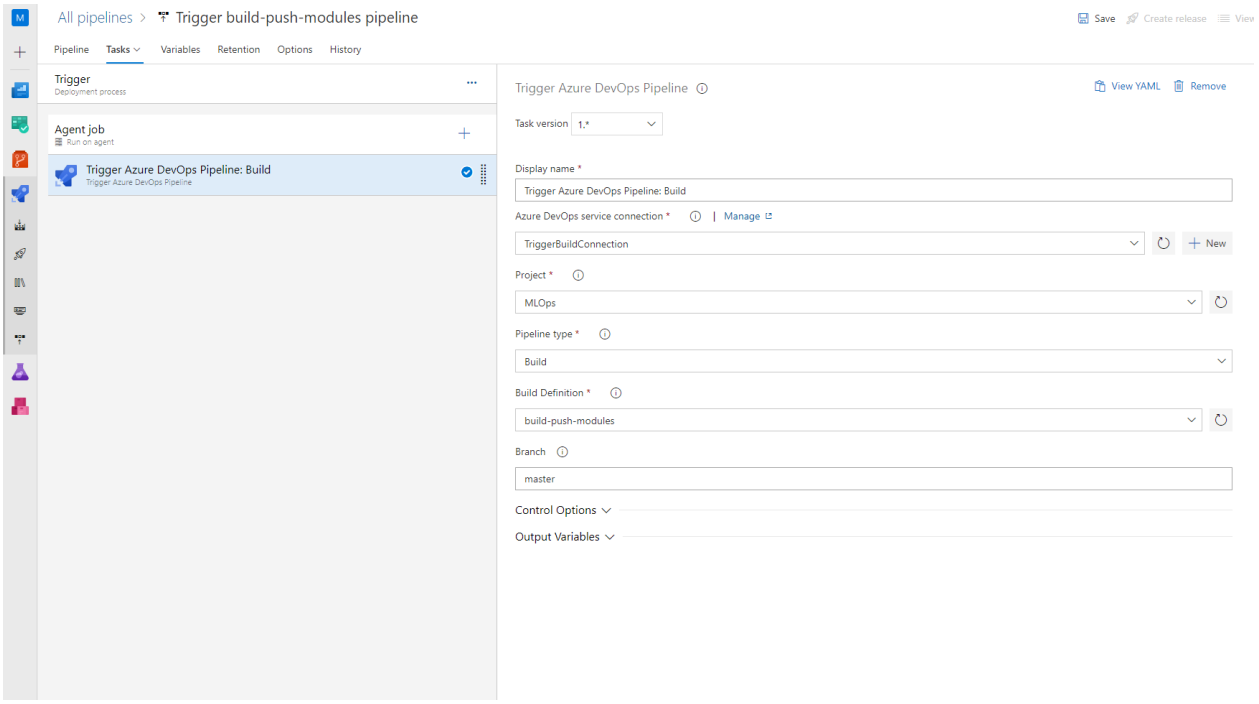
3. Create a new Release pipeline and set the artifact as an AzureML artifact, select the service connection you just created and for the name, give it the same name as the name of the model (here *yolo.onnx*)



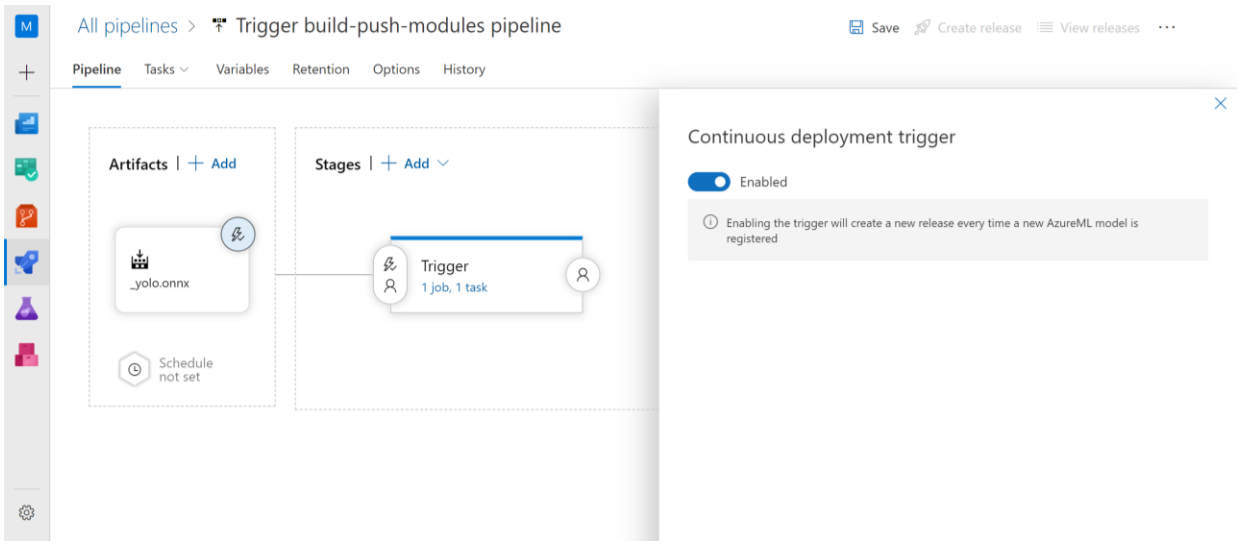
4. In the **Tasks** section, add a new **Trigger Azure DevOps Pipeline**.



5. Fill in the values you have set up so far and for the **Build Definition** field, give it the name of the build pipeline you are going to configure (you can call it `build-push-modules` for example)



6. Don't forget to enable continuous deployment by clicking on the lightning bolt icon next to the artifact

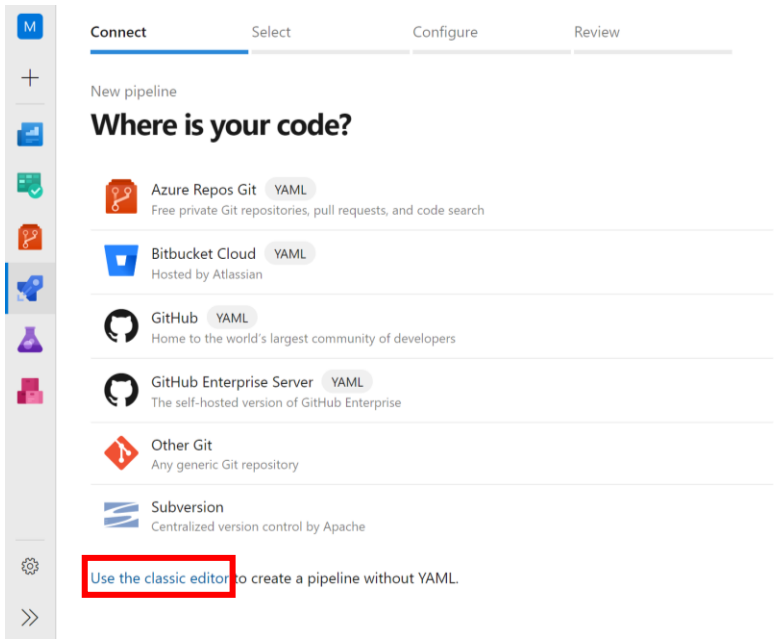


And that's it! All it does is to trigger a build pipeline but why do you need such an intermediate step?

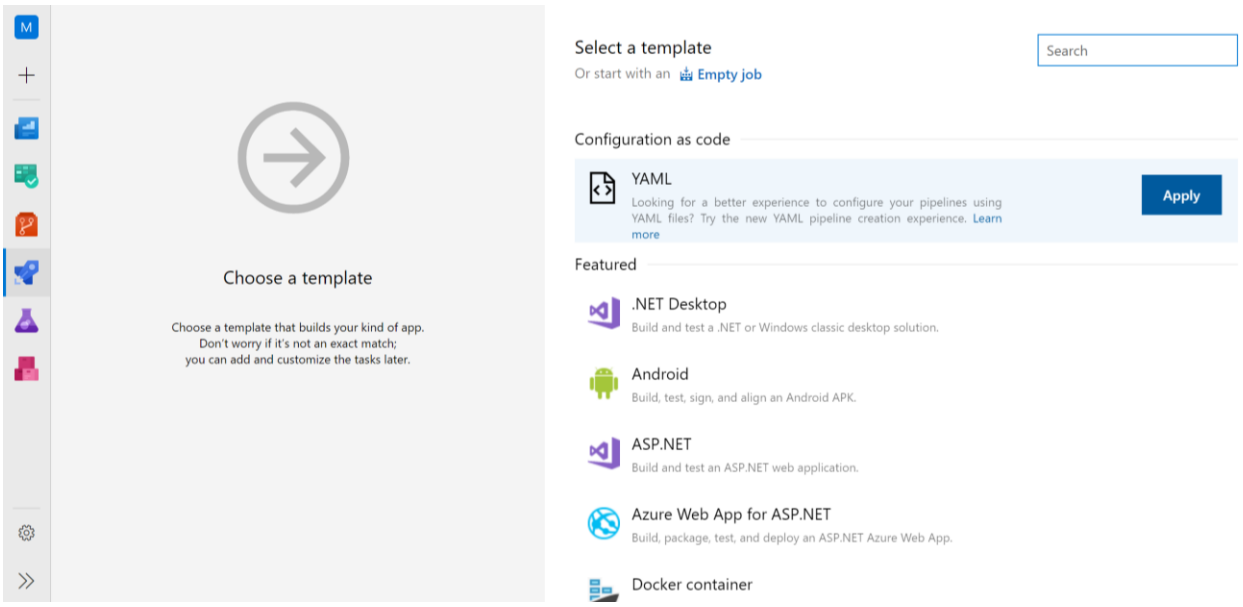
The problem was that you needed to have your source code (the *Rpi3-objectdetection* folder) in order to build and push your IoT Edge modules, so you can't build your modules in a Release pipeline - You would have to upload all your source code as an artifact, which is not appropriate in a fast-paced environment such as CI/CD -.

Now, let's configure the BUILD-PUSH-MODULES pipeline.

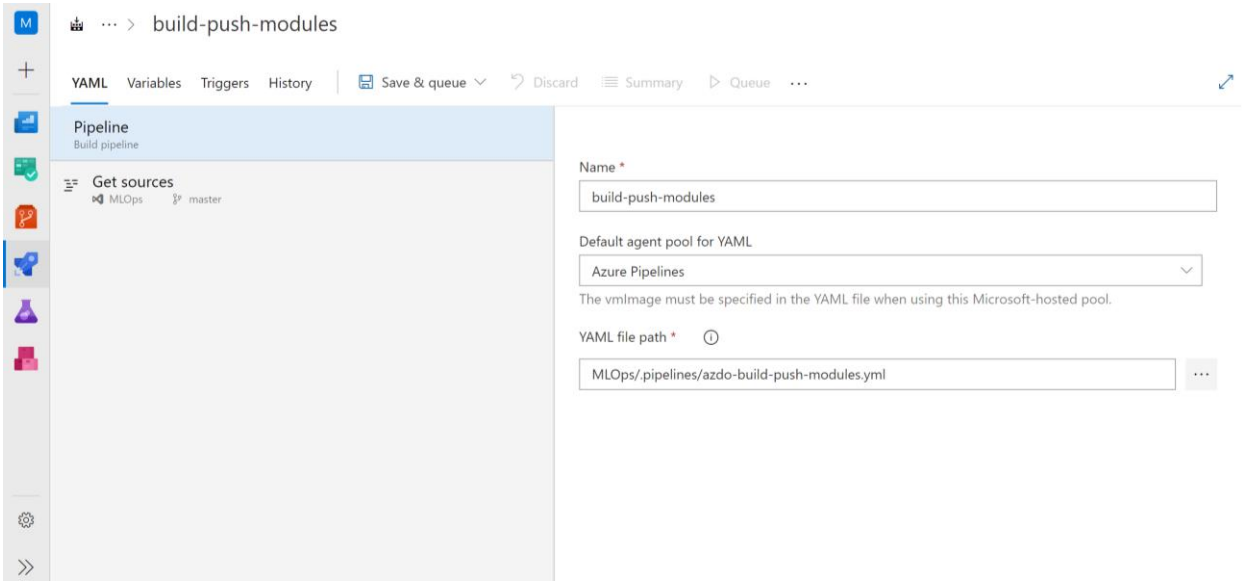
7. Create a new Build pipeline but this time, use the classic editor (since you need to give a name to the pipeline)



8. Select your repository and initialize a pipeline with the YAML template.

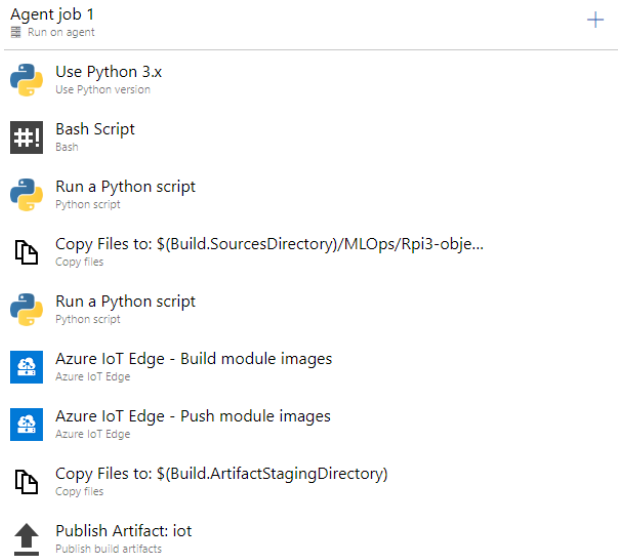


9. Give it the same name as the one you used for the trigger build pipeline (`build-push-modules`) and select the corresponding yaml file *azdo-build-push-modules.yml*



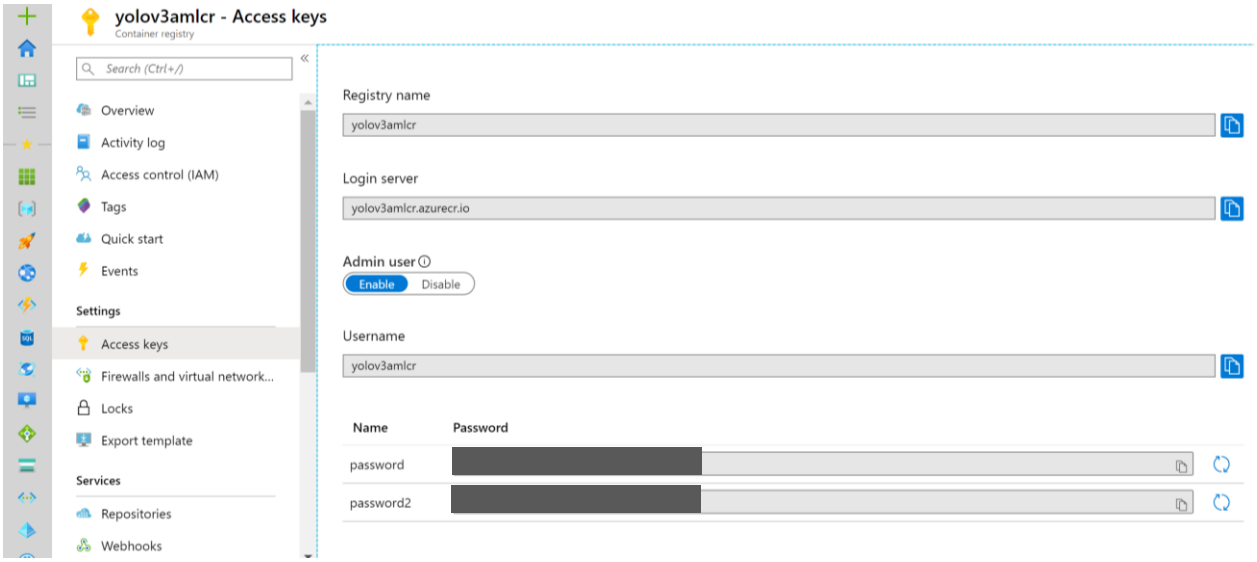
You don't need to run it because it's meant to be triggered, but if the training is already finished on Azure ML, you can manually trigger it to build/push the modules with the new registered model - It will always take the latest registered model -. This may take a couple of minutes/hours depending on your IoT solution.

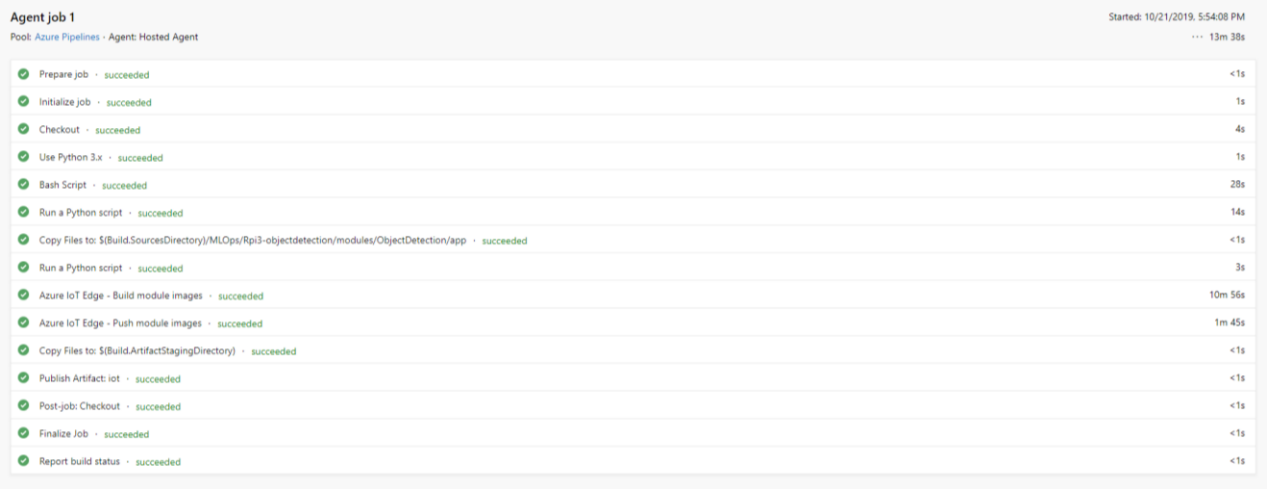
Here are the different steps it performs:



- The pipeline is alerted of a new registered model so it retrieves via a Python script the model from Azure ML and then copy it as well as the *classes.txt* file needed for the Computer Vision module to the corresponding folder (inside *Rpi3-objectdetection/modules/ObjectDetection/app*).
- It builds according to the deployment manifest the required modules (with Docker).
- Then pushes to Azure Container Registry the built docker images (via the env variables defined in the `iotedge-vg` variable group).
- It then publishes the *deployment.template.json* as well as the *module.json* files for the modules as a packaged artifact to act as the final trigger for the release pipeline.

Therefore, to make this pipeline work, you need to specify in the variable group `iotedge-vg` your ACR credentials (you can find them on Azure portal in the corresponding section):

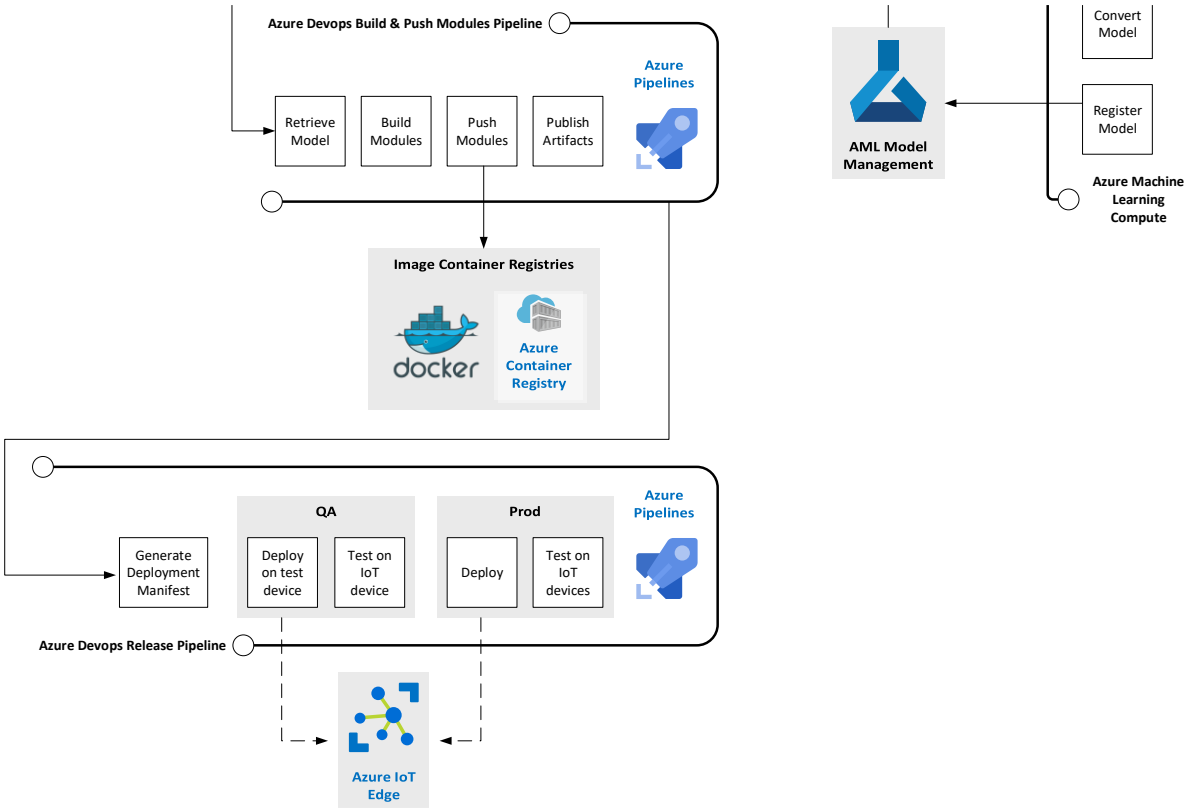




Once the pipeline finishes building and pushing the image, you are now left with the last pipeline, the release one.

Configuring the release pipeline

Apart from the intermediate "Release" pipeline (which was just intended to trigger a Build pipeline), there should only be in your projects a unique release pipeline so that you can easily manage releases and deployments (whether for the Edge or the Cloud).



In a release pipeline, there should be 3 stages (which corresponds to 3 different environments):

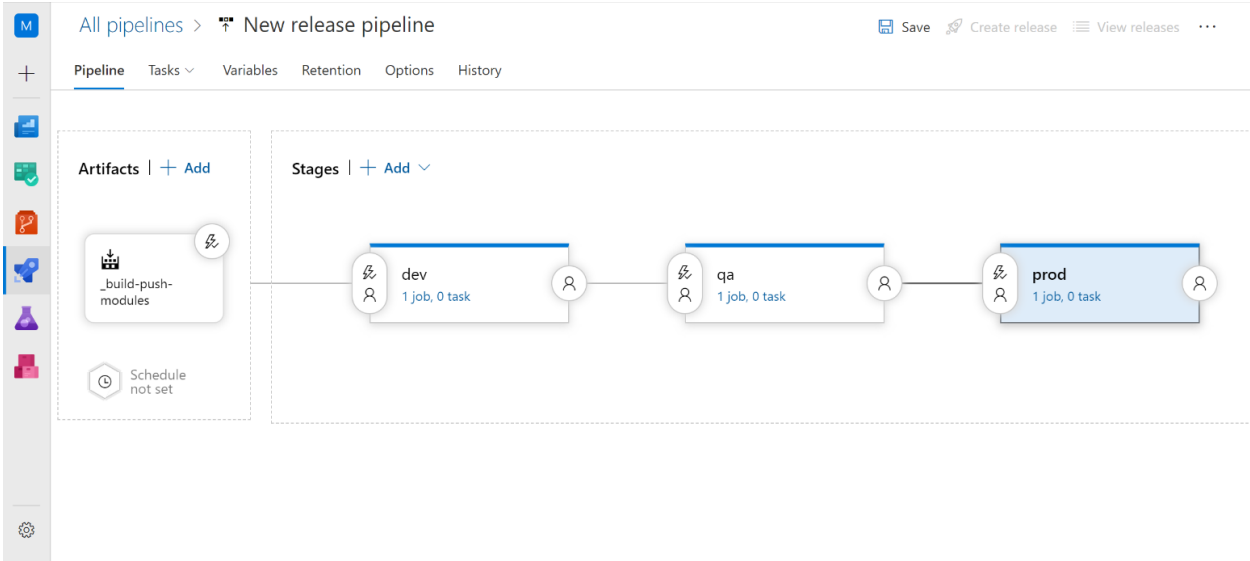
1. **Dev** (Development). This is the environment in which you and your team are working on for local testing, debugging, etc. This is where you work with your local devices
2. **QA** (Quality Assurance). This stage is intended to showcase or to test your project on a larger scale (typically, you can show a QA release to a client for a demo etc. you can also let another QA company review your product).
3. **Prod** (Production). This is the final stage where you deploy your solution to your clients (whether for an update or a complete overhaul of your current solution).

For each stage, you can define post-deployment approvals to ensure quality and stability before moving forward.

Here, you should already have tested in the past modules the solution (I hope!) so technically, you only need to define **QA** and **Prod** environments even though it is a good habit as a DevOps engineer to set up the whole release process.

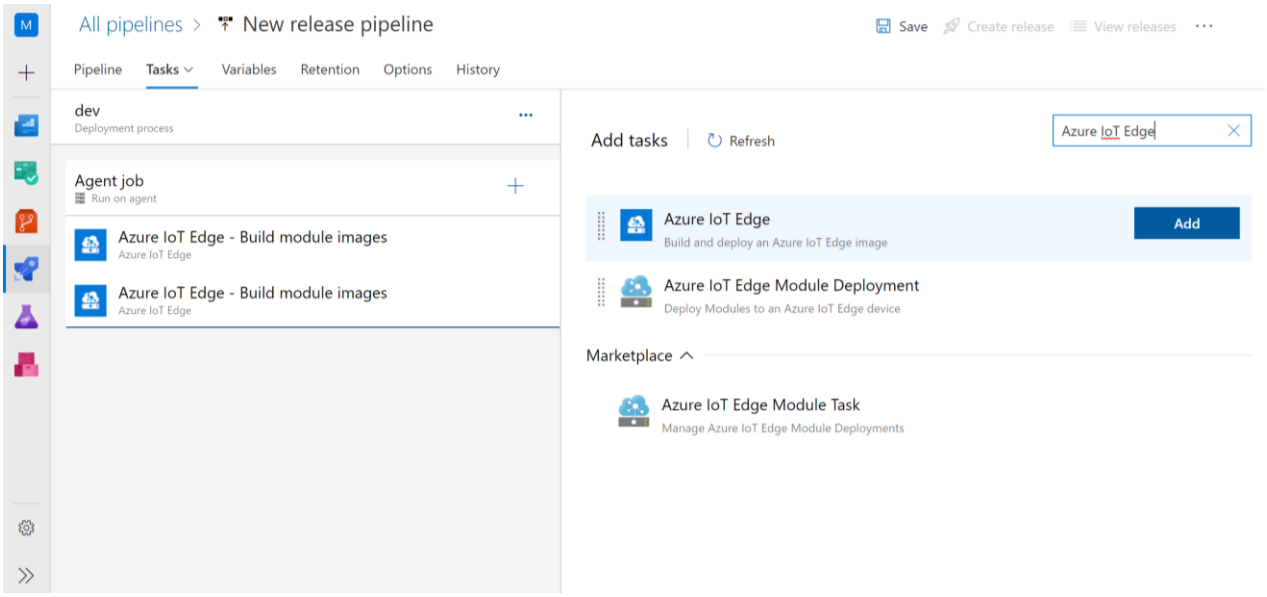
Let's jump in, to configure your release pipeline, perform the following steps :

1. Create a new Release pipeline and start with an empty job, enable continuous deployment, insert the artifact created previously (`build-push-modules`) and three stages **dev**, **qa** and **prod**.

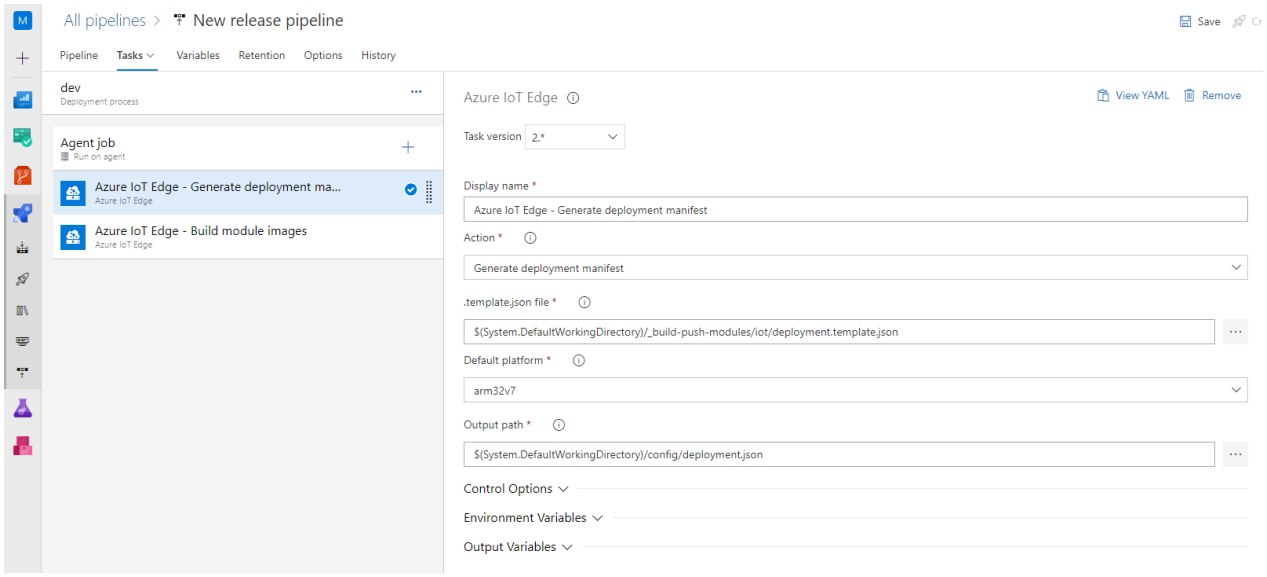


   These three stages are identical so let's go through the **dev** one.

2. Add an Ubuntu 18.04 agent and 2 Azure IoT Edge tasks.



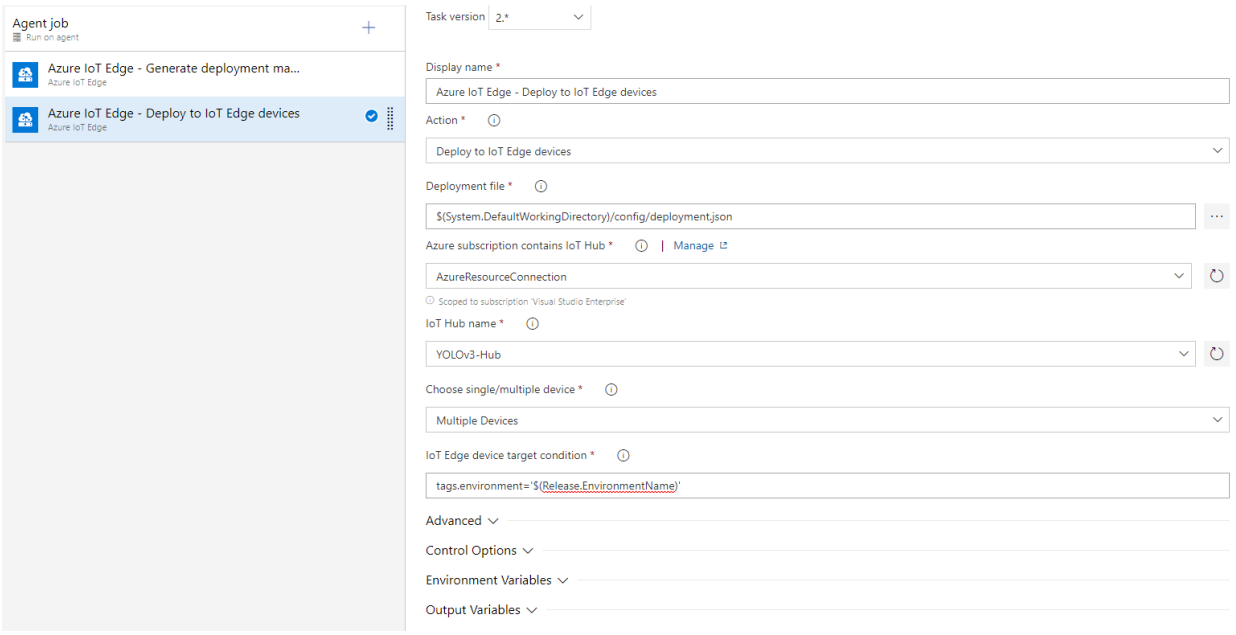
3. For the first one, select **Generate deployment manifest**, **arm32v7** for the default platform and specify the path to the *deployment.template.json* from the artifact.



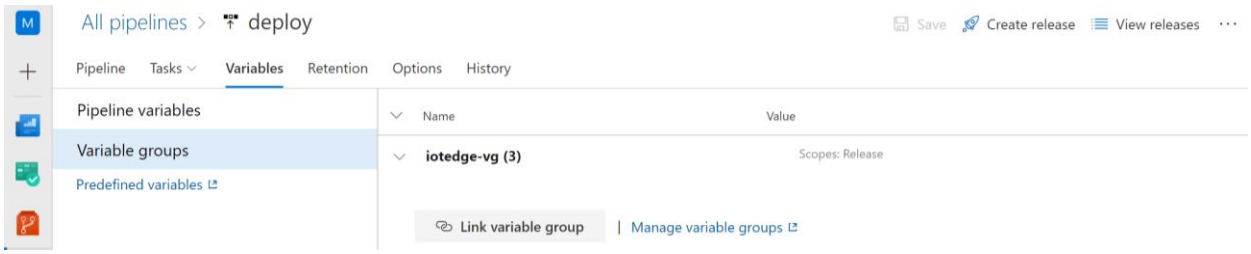
4. For the second one, select **Deploy to IoT Edge devices**, specify the values and for the target condition, fill in:

```
tags.environment='$(Release.EnvironmentName)'
```

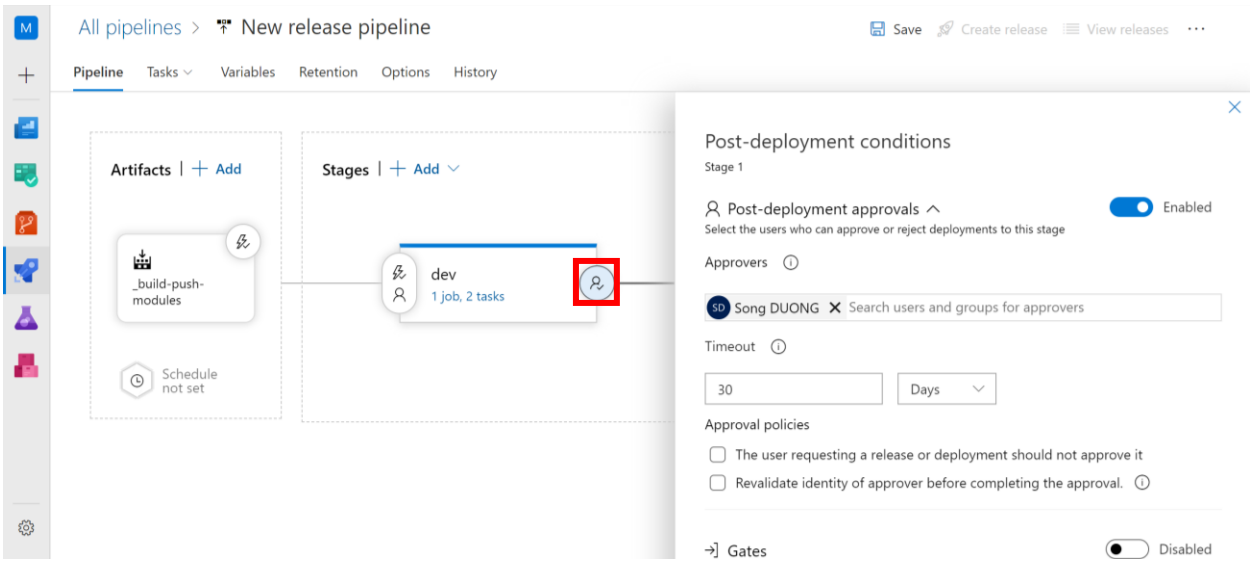
(as the release stage's name is 'dev', it will deploy to devices with 'dev' as tag so for 'qa' and 'prod', you should tag accordingly when registering your devices)



5. Link the `iotedge-vg` variable group to the pipeline to allow authentication to your ACR.



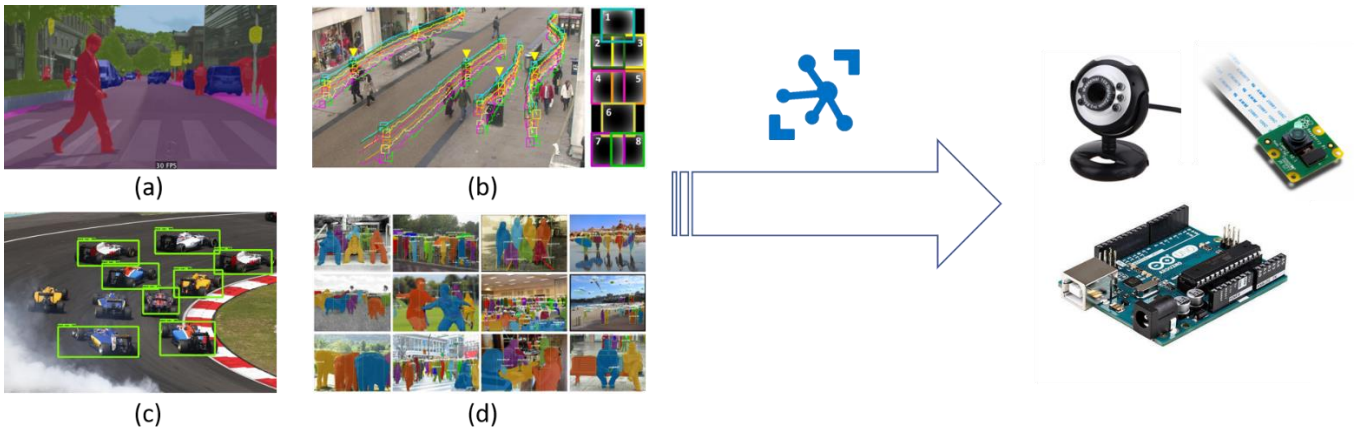
6. You can add post-deployment approvers by clicking on the icon on the right of the stage.



As a conclusion

After this guide, you should have learned how to develop from start to finish an IoT solution that can perform object detection on an edge device, here a tiny Raspberry Pi 3 Model B device.

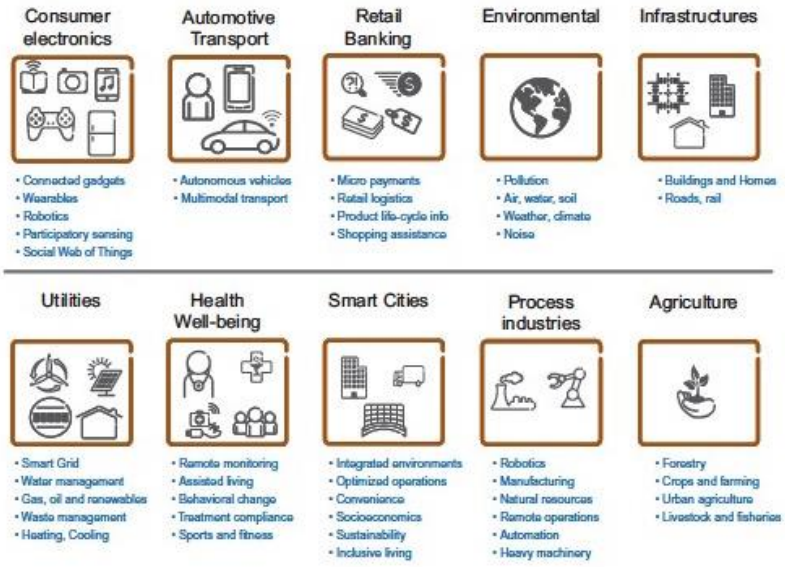
The "Intelligent Edge" is a promising area where many Computer Vision problems are yet to be implemented. By moving workloads to the Edge, IoT devices tackle today's big data challenge by processing data beforehand and reporting only the significant results, resulting in less interaction with the cloud and enabling devices to work even in environment that has intermittent access to the Internet and cloud, thus operators can rely on local resources exposed by the edge without worrying about their availability.



(a) semantic segmentation (b) object tracking (c) object detection (d) applications on Today's world

You should now have a wide overview of how to bring Computer Vision models to the Edge from the training of the model to the deployment on an edge device.

Not only you are able to rebuild the solution introduced in this guide, but you can also develop from scratch custom modules to add further layers of processing or filtering at ease. You can for instance easily integrate other Computer Vision models into your solution in the form of modules.



Computer Vision models illustrate very well the use of (I)IoT edge devices in today's world but there remains endless applications in other area that may or may not require a camera.

By dividing the problem into custom modules, Azure IoT Edge enables you to distribute work more efficiently among developers, boosting the creation of Proof of Concept to deliver quality products faster.

The webcast Deep Dive: Machine Learning on the Edge - Predictive Maintenance [102] available on the Azure IoT Show provides a complementary in terms of AI at the Edge (See also the related tutorial[103] and repo[104] on GitHub).

This concludes this guide for developers and data scientists.

[102] Deep Dive: Machine Learning on the Edge - Predictive Maintenance: https://channel9.msdn.com/Shows/Internet-of-Things-Show/Deep-Dive-Machine-Learning-on-theEdge-Predictive-Maintenance

[103] TUTORIAL: AN END-TO-END SOLUTION USING AZURE MACHINE LEARNING AND IOT EDGE: https://docs.microsoft.com/en-us/azure/iot-edge/tutorial-machine-learning-edge-01-intro

[104] IoTEdgeAndMlSample repo: https://github.com/Azure-Samples/IoTedgeAndMLSample

Appendix. Using Azure Notebooks

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

Some of the activities of the walkthrough are broken up across two notebooks.

1. *00-train-yolov3.ipynb*. This notebook walks through the steps to train and publish a model using Azure Machine Learning.
2. *01-configure-iot-hub.ipynb*. This notebook walks through the steps to create and configure your Azure IoT Hub.

The steps in this appendix might be typically performed by data scientists.

Setting up Azure Notebooks

Creating an Azure Notebooks account

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

Perform the following steps:

1. Navigate to Azure notebooks at https://notebooks.azure.com/.
2. Click **Sign In** in the upper, right-hand corner of the page.
3. Sign in with either your account.
4. If you have not used Azure Notebooks before, you will be prompted to grant access for the Azure Notebooks app.
5. Create a user ID for Azure Notebooks.

Uploading Jupyter notebooks files

In this step, you will create a new Azure Notebooks project and upload the Jupyter notebook files to it.

Create a new project and upload the files to your notebook.

1. Select **My Projects** from the top menu bar.
2. Select **+ New Project**. Provide a name and an ID. There is no need for the project to be public or to have a readme.
3. Select **Upload** and choose **From Computer**.
4. Select **Choose files**.
5. Select all the guide's Jupyter notebook files and click **Open**.

6. Select **Upload** to begin uploading and then select **Done** once the process is complete.

Running Azure Notebooks

Now that the project is created, run the *00-train-yolov3.ipynb* or *01-configure-iot-hub.ipynb* notebook as per module.

Perform the following steps:

1. From the turbofan project page, select the Jupyter notebook .ipynb file.
2. If prompted, choose the Python 3.6 Kernel from the dialog and select **Set Kernel**.
3. If the notebook is listed as **Not Trusted**, click on the **Not Trusted** widget in the top right of the notebook. When the dialog comes up, select **Trust**.
4. Follow the instructions in the notebook.
    - `Ctrl + Enter` runs a cell.
    - `Shift + Enter` runs a cell and navigates to the next cell.
    - When a cell is running, it has an asterisk between the square brackets, like **[*]**. When it is complete, the asterisk will be replaced with a number and relevant output may be appear below. Since cells often build on the work of the previous ones, only one cell can run at a time.
5. When you have finished running the Jupyter notebook, return to the project page.

Microsoft France
39 Quai du Président Roosevelt
92130 Issy-Les-Moulineaux