4.1 - MLOps: Pipeline Controlling

Professionalising CI/CD pipelines

The pipelines for MLOps are a spin-off of DevOps pipelines where we are trying to implement the best practices of CI/CD. CI/CD is split up into two parts:

- CI - Continuous Integration
- CD - Continuous Deployment / Continuous Delivery

The difference between the two aspects of Deployment and Delivery can be found in the automation aspect: Continuous Deployment will automatically deploy into production whereas Delivery will set it ready to be easily deployed with a few simple clicks.

Many of the CI/CD pipelines that are professionally used are set up in such a way, that a failure doesn't mean the pipeline should start from the beginning again. Also, things that haven't changed shouldn't be done again if that's not needed.

These aspects require the pipelines to be set up a little different than simply chaining scripts one behind the other. That's also a reason for using dedicated tools such as GitHub Actions as our CI/CD tool.

For MLOps, this isn't different, we need more control on the pipeline. What is going to run when? How do we react when only a part of the pipeline needs to be executed again? How are we going to store artefacts in between? ...

Most of this, such as the caching, is already handled by Azure, so that components that have a similar **Input, Code, Output,** will be cached and do not need re-submitting.

Below follows some more theory behind setting up pipeline controlling and traceability in our MLOps pipelines.

Pipeline controlling

As we mentioned in one of the theoretical sessions, and you might have noticed yourself already, it is not always useful to run our pipeline from top to bottom.
Sometimes we want to skip a few steps in order to speed up the pipeline and we don't have to run the earlier steps of the pipeline when they're already done before.

E.g.: *When we only want to execute a new training job, we want to skip our Data processing pipeline.*

Luckily, a few of these controls have already been built in into Azure ML Studio nowadays. When Azure detects that there are no changes to the components, or the datasets, or the code, they are just cached and they use the version that has previously succeeded. That way the pipelines run smoother and faster.

However, there are a few other steps you can easily skip, or configure:

- `create_compute`
  Do we still need to create a compute machine, or can we just use the one we already have?
- `train_model`
  Set this to `False` if you want to test all the rest of the job **steps**, except for the actual Azure model training pipeline
- `skip_training_pipeline`
  If you set this to `True` we can skip the whole Azure part, so we can go directly to the next **job**

- `deploy_model`
  This helps you in allowing the pipeline to deploy the model into Kubernetes or not. If you toggle this off, you’re just training and not deploying.

When running the GitHub Actions workflow using the `workflow_dispatch` you can use checkboxes for the boolean values. We will be configuring that later on.

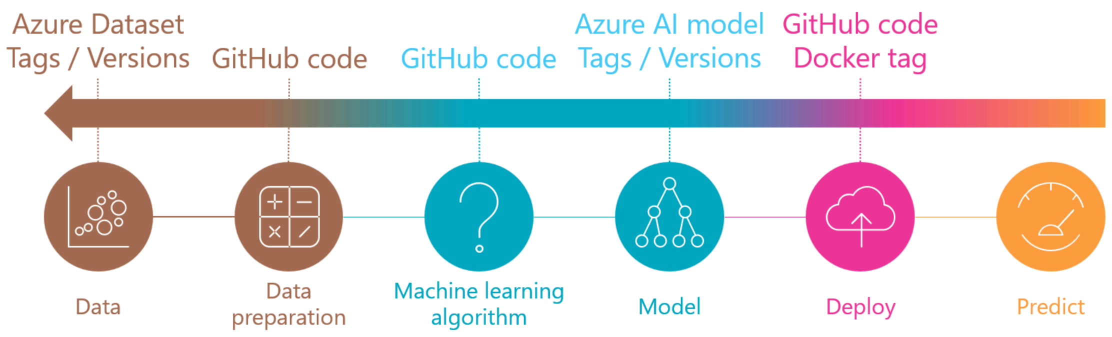
🚨 The first time you're running the pipeline, you will need to run everything starting from the beginning of the pipeline, otherwise we haven't done any Data Processing.
This is the so called **cold-start** and is very important to also test!

Version controlling in MLOps pipelines

In Figure 1 below we can see some difficulties regarding Version Control throughout the MLOps Pipeline. There are lots of different Version Controlling options for each different step of the project.
What we want to do is find out all the information regarding different versions used in our AI Project.

Imagine the scenario where a user is complaining about a wrong prediction, or a misclassified picture. The incorrect classification *should* be logged together with as much information about the deployed project as possible. It is not that hard to find out what version of the API the user was working on, as a developer could've linked it in the Footer somewhere. Let's say he used **Version 1.2.3** of the API.

Now, the difficulties are encountered when we want to find out which AI model was deployed into API **version 1.2.3**. We will go from right to left to find out what change was made that broke this API. This is not something we have already seen in the previous topics, but it is very important to think about.


Azure Dataset Tags / Versions
GitHub code
GitHub code
Azure AI model Tags / Versions
GitHub code Docker tag
Data
Data preparation
Machine learning algorithm
Model
Deploy
Predict
Figure 1: Version Controlling MLOps Pipelines. Own illustration.

One important thing we can use during a lot of these steps is working with GitHub. GitHub allows us to easily **link changes to a user**, as a user's commits are always generating a **SHA-1 ID.** How is the GitHub SHA generated? – Stackoverflow.com.
This Commit ID, often referred to as the SHA, is an ID that is unique to the repository, and allows you to identify who made what change and when.
Using this information, we can easily blame users when something went wrong at a certain point.

Check out these references:

- Git Blame – Git SCM
- Git Blame – Bitbucket

Let's try to implement the GitHub SHA in our visualisation

Data

If we start on the left side of the pipeline in Figure 1, we can see our AI Data.
This has been **labelled** with an **Azure Dataset**. In the assignments, we saw that it was possible to attach **Tags** or **Versions** to our Azure Datasets. When we are going to label our data in Azure, we can give it specific versions.
We don't have to link it to our **GitHub SHA**, as the raw data is not always going to change by code.
You could link it to specific SQL queries if you are creating a Database export and uploading that, if you want to.

Data preparing

Going on to the next step, our Data preparing code, we are going to link our **GitHub SHA** to the prepped data. We do this because our Data preparing script is included into GitHub and can be a reason why the AI model gives wrong results. E.g.: The images are not correctly resized. OpenCV (or other imaging processing libraries) are given the wrong results after an update ...

This data preprocessing script is executed in the Azure Components, and through that immediately version tagged. The cool thing is that it automatically registers the GitHub SHA as metadata to the component, when it's registered from a GitHub Pipeline. This is very interesting for retracing purposes.

AI Training

The following step, the AI training, can be Version Controlled simply by using GitHub and the default Azure Machine Learning tools. It's a similar way as the data preparing.

We want to keep track of the structure of our AI model, and the code to recreate that. In our previous assignment, all of this information resided in the `train.py` script. In our MLOps project, this will be a file that exists in GitHub, and is thus linked to our GitHub SHA.

Remember that Azure automatically keeps a Snapshot of the Component's scripts? Well, this comes in play now! The Snapshot information is already been stored, so that all the code can be found in the Azure Machine Learning Workspace. It's also linking back to the used **Training and Testing data** in the Azure Machine Learning Workspace.

All of the metrics that come out of this AI model training job are also kept, which can be very useful for later jobs.

Model saving

Next comes the saving of our actual trained AI model. Azure links this to our Training Job as this is a direct result of such a training job. This AI model can also be organised with Tags such as the GitHub SHA.

We implemented it in such a way that the AI model is automatically saved and stored in the Components outputs, and such also uploaded into the Blob storage.

Model packaging (Docker Image)

Finally, our AI model will be **downloaded**, combined and packaged with our API code into a Docker Image. This is by far the easiest way of version controlling everything. Simply by adding a new Docker Tag, we can specify the GitHub SHA and organising the Docker Tags in that way. Tagging Docker image the right way – container-solutions.com.

This doesn't stop you from using the Semantic Versioning (v1.2.3) in your Docker image as well... Docker Images are also hashed in a similar way as GitHub Commits. When you label a Docker image with multiple labels, they refer to the exact same version. In the table below, you can see that there are bunch of Docker images with different tags, each having the same Image ID and are thus exactly equal.

```
REPOSITORY                          TAG                                        IMAGE ID       CR
EATED       SIZE
ghcr.io/nathansegers/azure-mlops-api   05ee055698af2171294615a1e423f
71f4098628e   3f49a154a09c   5 days ago     998MB
ghcr.io/nathansegers/azure-mlops-api   05ee055                                    3f49
a154a09c   5 days ago     998MB
ghcr.io/nathansegers/azure-mlops-api   customer-release                           3
f49a154a09c   5 days ago     998MB
ghcr.io/nathansegers/azure-mlops-api   latest                                  3f49a1
54a09c   5 days ago     998MB
ghcr.io/nathansegers/azure-mlops-api   v1.2.3                                  3f49a1
54a09c   5 days ago     998MB

## These images are uploaded to Docker Hub instead of GitHub Container
Registry
nathansegers/azure-mlops-api         v1.2.3                                3f49a154
a09c   5 days ago     998MB
nathansegers/azure-mlops-api         05ee055698af2171294615a1e423f71f
4098628e   3f49a154a09c   5 days ago     998MB
```

What about skipping and caching?

Sometimes it is possible that earlier steps of your MLOps pipeline do not need to be run again. Parts of the pipeline where **Input** and **Code** are not altered, will simply be cached for the next time. That way, we don't need to waste time to run the code if the result will be the same. This is the reason why the components should be configured as `is_deterministic` . This setting tells Azure that the Output will be the same if the Input remains the same. If you're working with Random settings which don't have a fixed Seed, they will be randomised and return different results in every run.

This part of the **Caching** in Azure is covered quite easily. GitHub Actions also helps is with some of the caching parts, if you're keeping track of **Artefacts**. Those are a little bit harder to set up, but will give you quite some flexibility in the pipeline if configured correctly.

Another part of flexible pipelines will be to hop over to specific jobs immediately. We can actually skip certain parts of the pipeline, similar to the caching strategies. By saving results in-between Jobs in the pipeline, we can easily pick up where we left off.

This is demonstrated clearly in the deployment of the AI model, which can be downloaded from Azure. As the model is stored in the Blob storage on Azure, it can be downloaded using one of the many `azcli` commands. A GitHub Actions Runner can get if from the cloud storage, put it in the source code of an API, and then package it all in a Docker Image. This step is often repeated if the API code is changed, but the AI model is not. During development the API changes a lot, and time is precious.

Using the right version tagging is crucial for all of the flexibility functions we want to use.

Summarise how to find information

Let's summarise how we can get all the information now.

1. **Find out who changed the API Code**

Imagine you have your API deployed on Kubernetes, and you want to find out who did the latest change.
Most likely, you have kept some kind of metadata in the API. This can be the version you are using, like `v1.2.3`.

```
API Metadata → ENVIRONMENT VARIABLE (API VERSION) → v1.2.3

v1.2.3 → DOCKER IMAGE ID → 05ee055698af2171294615a1e423f71f4098
628e

05ee055698af2171294615a1e423f71f4098628e → GitHub SHA → User In
formation + Code
```

2. **Find out which AI model was used in the API Code**

We find it's best to link a reference to the AI model version in an environment variable of the API code. This way you can always see which AI model was deployed in your API.

```
API CODE → ENVIRONMENT VARIABLE (AI MODEL) → v1.0.0

v1.0.0 → AZURE MACHINE LEARNING WORKSPACE → Training Job/Exper
iment information

Training Job Information → TAGS → GitHub SHA

GitHub SHA → User Information + Code
```

3. **Find out which AI data was used in the API Code**

You can delve deeper into the Azure Machine Learning Service to find information regarding the data that has been used to train our AI model.

```
AI MODEL v1.0.0 → AZURE MACHINE LEARNING WORKSPACE → Training
and Test Set

Training and Test Set → TAGS → GitHub SHA
```

GitHub SHA → User Information + Code