COM774 – Practical 6 – CI/CD with ML and Azure

Azure CLI

We will be using the Azure CLI during this practical. You will need to install it onto the lab machines. You can find instructions here: https://learn.microsoft.com/en-us/cli/azure/install-azure-cli-windows?tabs=azure-cli

You want the latest 64-bit version. Once it is installed open powershell and use the "`az login`" command to log into your Azure account. Once that is installed we need to install the Azure ML extension with "`az extension add --name ml`". That will take a minute or two, while that is installing you can move on with the practical.

Azure machine learning

First, run through the setup process for our Azure ML workspace. But this time where it says "Container registry", create a standard one.

Once you've done that add your dataset again, and add your compute cluster (remember to go for the cheapest one).

Create a job

Now, lets create a job to build our model just to confirm it's up and running. The code for this week is on blackboard this time.

As in the last practical, go to the code directory where you have downloaded it and run the job.
You may need to make changes depending on how you have named your compute cluster etc.

Recap of where we are so far

In the first few practical's we got up and running with python, making a simple machine learning model for the iris dataset.

Moving forward from that we exported the notebook and used it to produce more production-like code which we then tracked with Git to monitor our changes and ensure we have traceability for what we're doing as well as records of what changes were made and why.

Next we got up and running with Azure ML, which let us track our jobs and link them to datasets for reproducibility.

We also worked through registering our models produced by the jobs and deploying them to endpoints we can use in our applications to get predictions from the model.

We're a good part of the way there from an end to end process. But there is still a gap in the middle between Git &our code, and Azure ML itself. Today we're going to look at closing that gap with Githubs CI/CD

GitHub actions

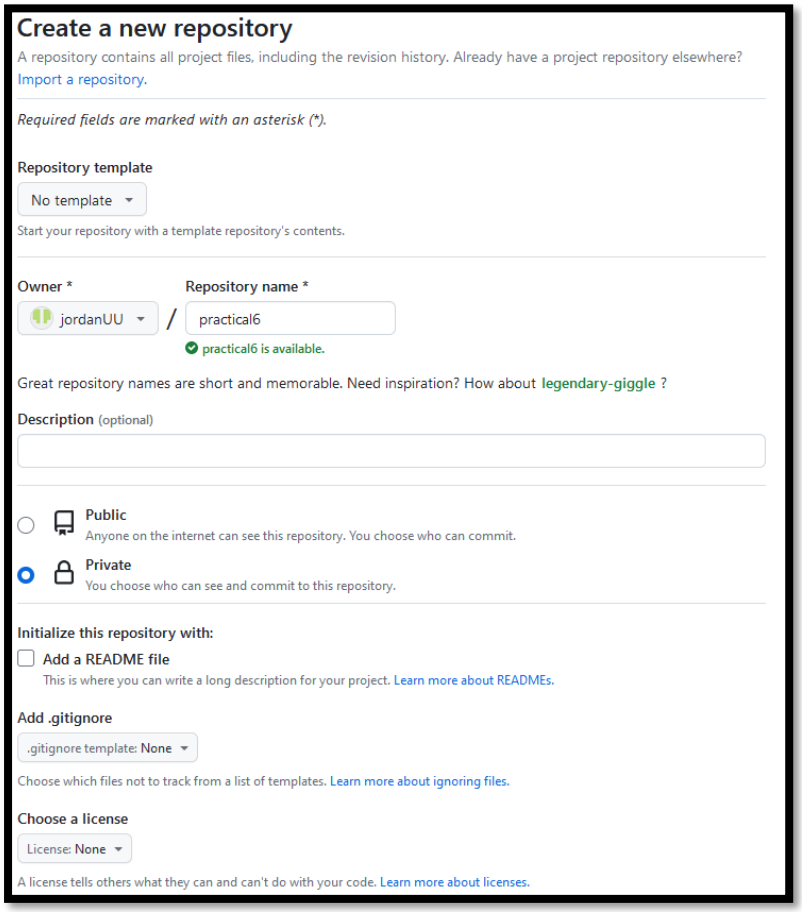
GitHubs CI/CD tool, which it calls actions, and CI/CD pipelines in general, are hopefully fairly straightforward. They are a series of instructions for automated CI/CD tools that tells them what we need them to do and under what circumstances we want them to do it. There is a few different steps to these.

1. Tell it when to do something – is it when you push a button, when you change something, when you merge something?
2. If you remember back to our docker information the first thing we had to tell it was “where to start”. CI/CD pipelines are similar. There is a lot of different potential operating systems and environments they could need to work from. Therefore we have to tell it where to start.
3. The next thing we need to do then is tell it what to do in that environment. In our case if we’re looking to submit jobs to Azure we need to instruct it to do that on our behalf.

Create your own repository

This week because we’re using CI/CD pipelines you’ll need extra permissions on the repository than you get with the classroom. So you’re going to need to make your own repository.

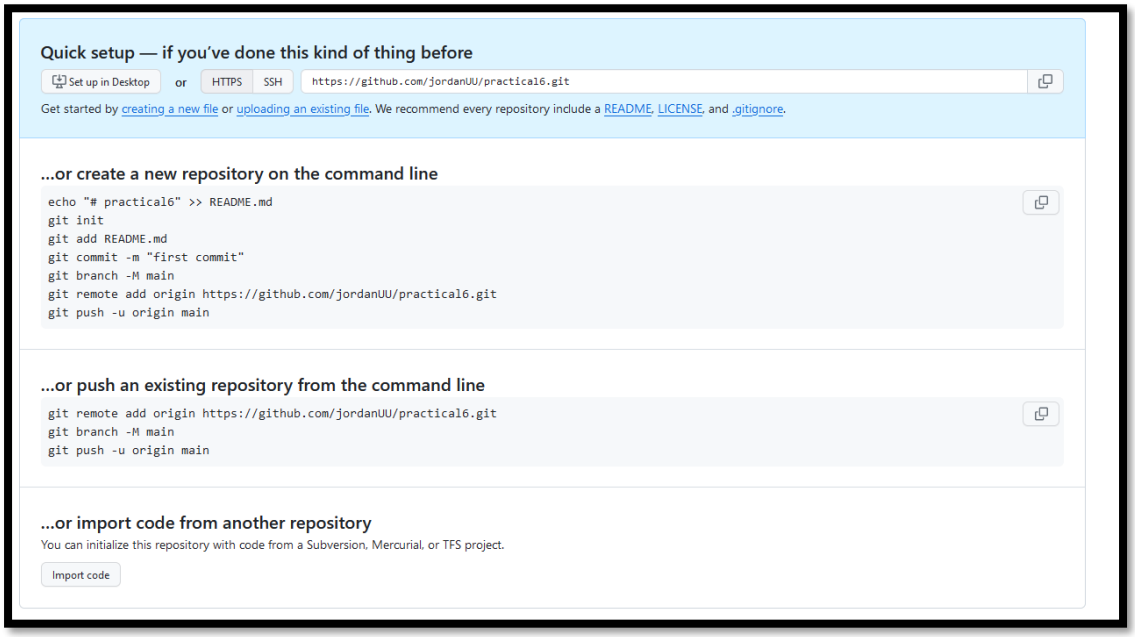
Go to GitHub and on the left beside repositories click “New” and fill in the details.



Then click create repository.

You'll be presented with instructions for different options. We already have code but we want to create a repository for it. Follow the first box for "create a new repository on the command line"

Use Git Bash as before, go to the place you have the repository code with "cd" and follow the instructions for GitHub.



Once that's done you should be able to refresh your screen and see your code. If you do, you're good to move on to the next step.

Testing Github actions

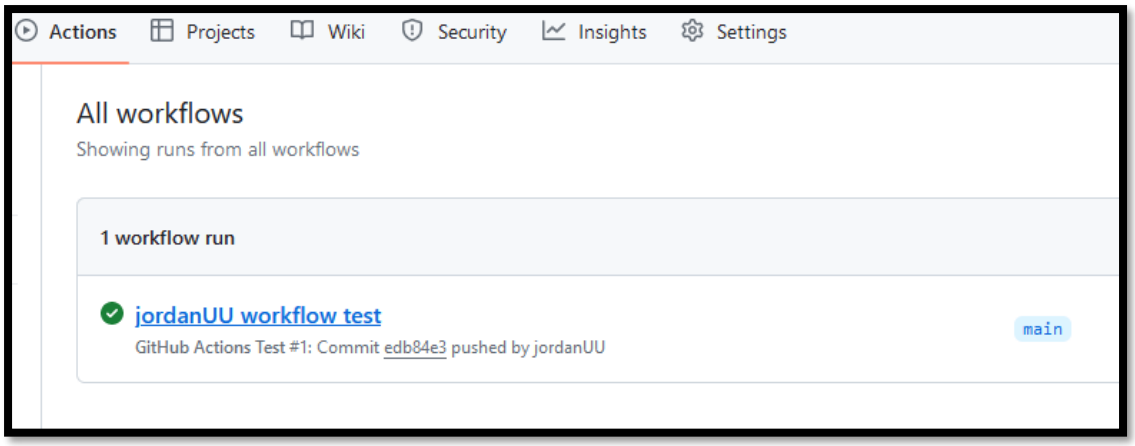
GitHub stores it's settings in a ".github" folder. GitHub often calls its CI/CD scripts "workflows" and therefore it looks for them in a ".github/workflows" folder in the repository.

I have a test action for you I've made in the code for this week. It's set to print out some basic information and to do so every time you change something. Take a quick look at it and make sure you understand what it's doing. It's similar to our Azure ML YAML files, with a slightly different syntax. But you can see we can still replace parts of it with variables in "${{ }}" to give our workflows flexibility.

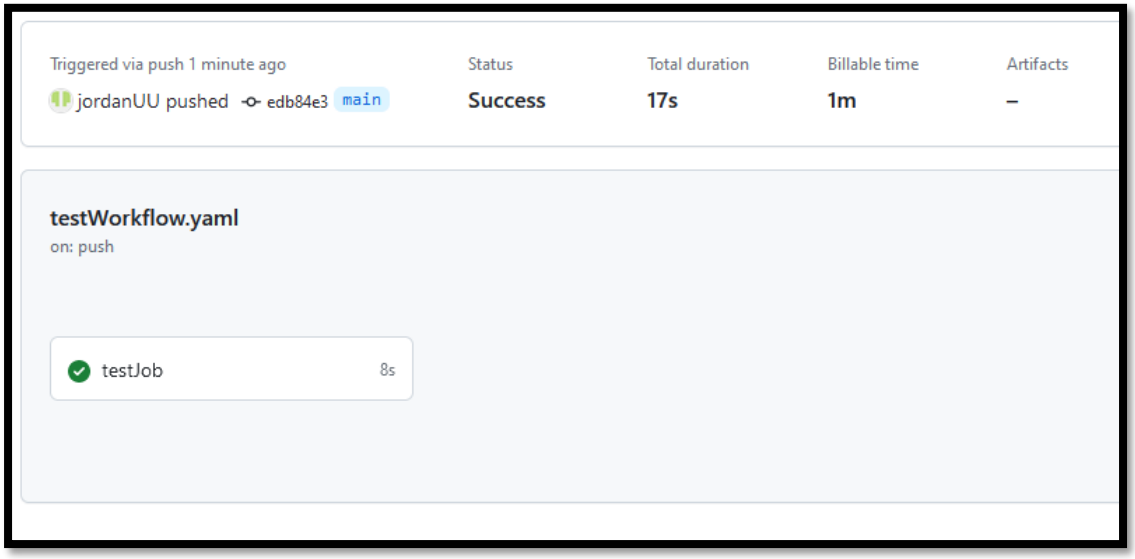
You can find some official guides for GitHub workflows at: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

Spend a little time reading that page so you understand what you're about to get into.

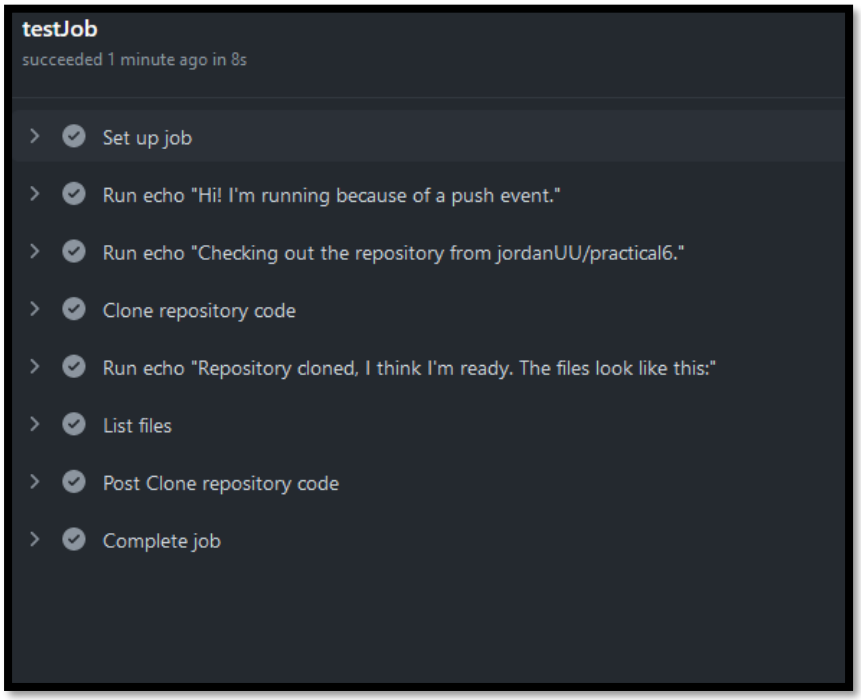
Now you've learnt a little about them testing this should be easy. Change the model to a different one, any one you want. Commit the file and then push it back up to GitHub. Once that's done go to the "Actions" tab and you should see the test workflow I made has run.



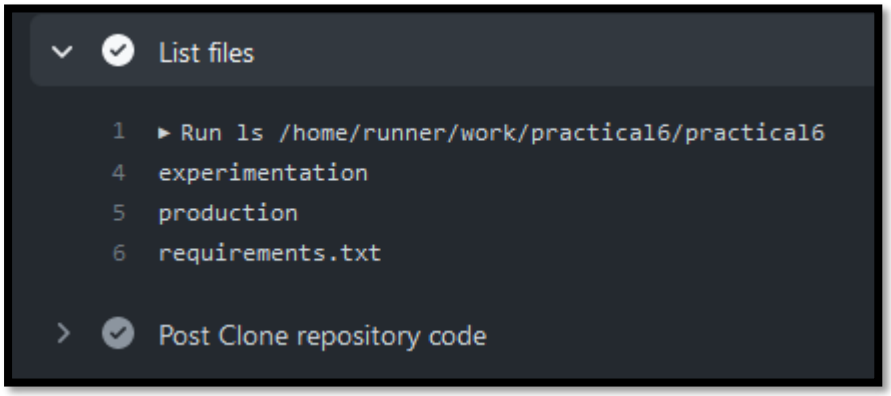
Within the workflow you can see the individual jobs and their status.



Inside of which again you can see the individual steps



We can expand one such as the “ls” step to see the result.



And we have a list of our repository files, showing it did indeed manage to clone the repository and download everything it needed.

Looks like we’re good to go onto the next step.

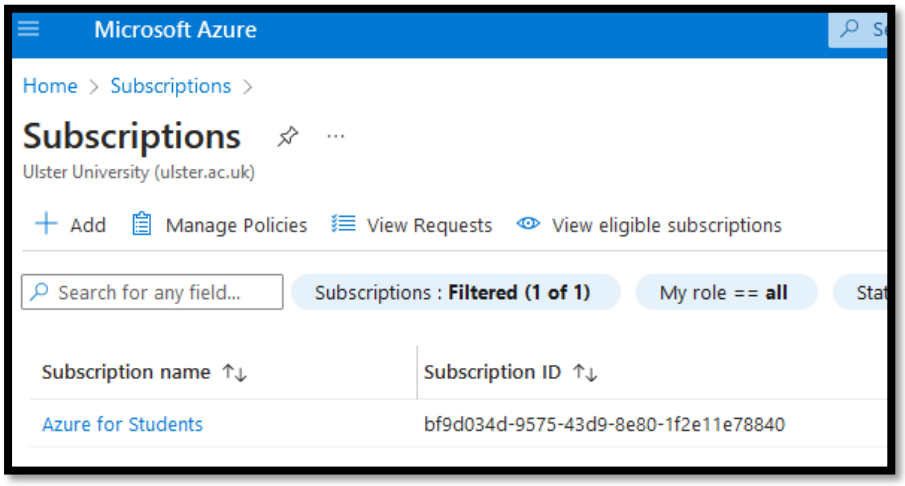
Security and Service Principals

So far, so good. Now we have the ability to get a worker up and running in our CI/CD system executing commands we need it to in order to automate some things for us.

As we know from submitting jobs using the Azure command line the first thing we have to do is log in. Now, this is going to be a shared repository people on the different parts of the team can commit to. We don’t really want them all using our account in case they do something… silly.

We need to make a shared credential that has access to our Azure machine learning workspace and can submit jobs. That way if we ever decide we don't want it to have access anymore we can just get rid of it without affecting our account. Azure calls these "Service Principals". They're essentially just accounts we can attach permissions to that are designed for automated usage and not generally for use directly by people.

We can create these with the Azure CLI, they're tied into our account, tenant (Ulster University) and subscription. So we'll need to get our subscription ID. Type "subscriptions" in the main Azure search box to search for the subscriptions view. Select it and you should see yours.



Copy it, you'll need it in a minute.

Now we have the subscription id we can create a service principal. Service principals have a "role" which defines their permissions. In our case we want to assign it "contributor" which gives it quite a lot of control.

We can create a service principal in the command line with the following command. Make sure to replace the parts in "<XXX>" with your own details:

```
az ad sp create-for-rbac --name "<your-bcode>-practical6" --role contributor --scopes /subscriptions/<subscription-id>/resourceGroups/<group-name> --sdk-auth
```

For me this looks like:

```
az ad sp create-for-rbac --name "e16001416-practical6" --role contributor --scopes /subscriptions/bf9d034d-9575-43d9-8e80-1f2e11e78840/resourceGroups/practical6 --sdk-auth
```

Yours will be a bit different

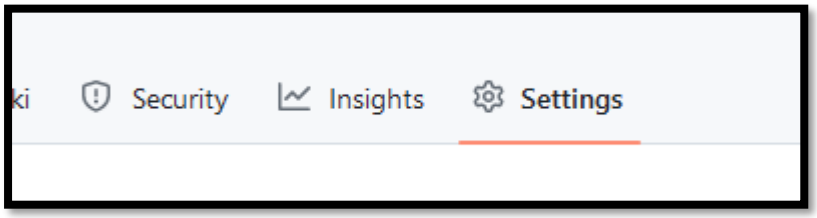
That's going to print out some details to your screen formatted like the following:

```
{
    "clientId": "<GUID>",
    "clientSecret": "<GUID>",
    "subscriptionId": "<GUID>",
    "tenantId": "<GUID>",
    (...)
}
```

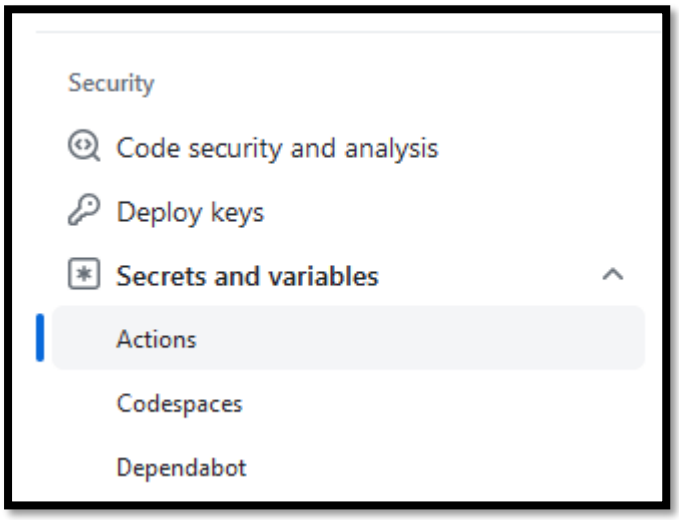
Save that entire block. You will need it shortly and you cannot get the client secrets again without resetting them. For security reasons.

That block is all the details we need to connect to connect to Azure as the service principal account we've just created – which has access to our whole resource group for this practical!

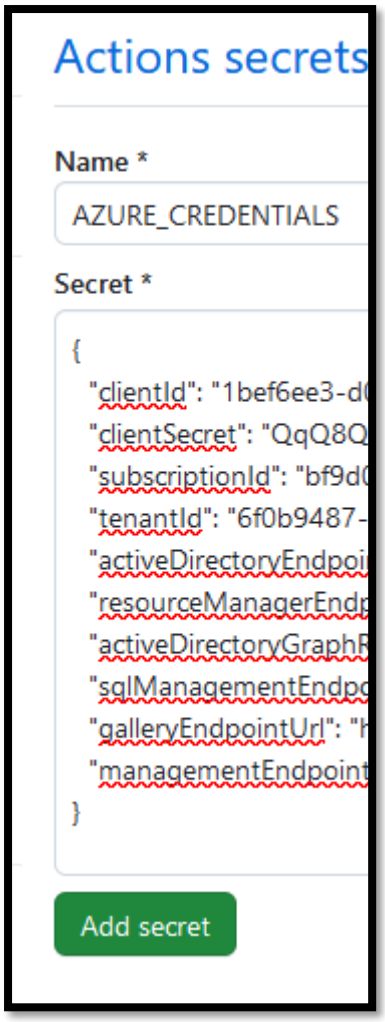
Now we need to give those to GitHub. Go to the repository page on GitHub and go to Settings.



Then under "Secrets and variables" select "Actions".



You'll see a "Secrets" and "Variables" tab. Make sure you're on "Secrets" (it should be the default). Select "New Repository secret" and call it `AZURE_CREDENTIALS`. Then put in **all** of that block you were given when you created the service principal.



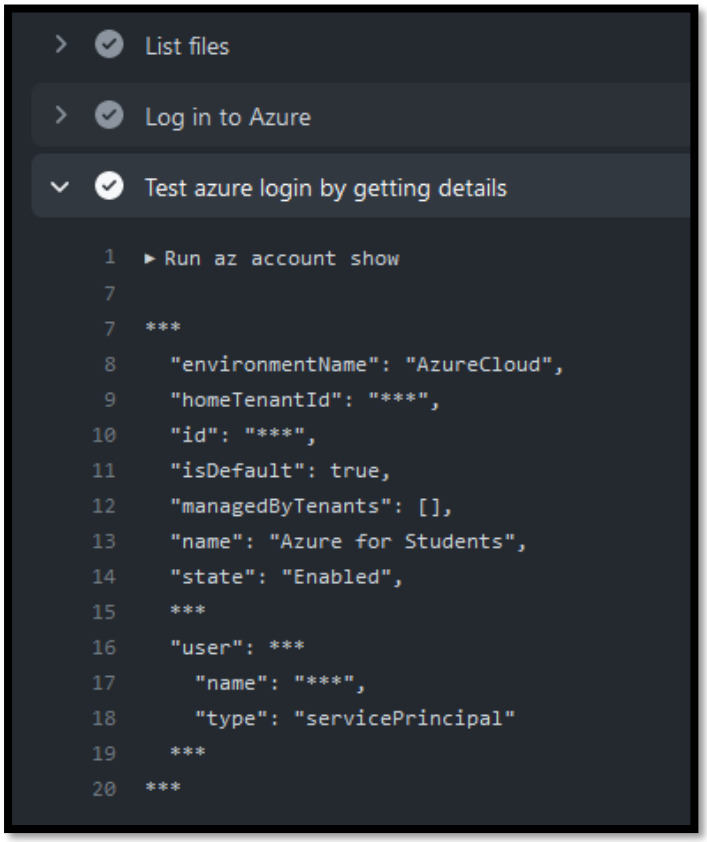
That's us most of the way there! The final step is we need to tell our GitHub action to log into the Azure CLI with those details. GitHub has a built-in for this called "azure.login@v1" which we can use like this:

```
      - name: List files
        run: |
          ls ${{ github.workspace }}
      - name: Log in to Azure
        uses: azure/login@v1
        with:
          creds: '${{ secrets.AZURE_CREDENTIALS }}'
```

Add that into the workflow file. Now we have a logged in azure CLI inside our worker. The next sensible thing to do is probably test that. We can show our current account details in the Azure CLI with "az account show" so lets run that and see what happens. Add that into the workflow too.

```
      run: |
        ls ${{ github.workspace }}
    - name: Log in to Azure
      uses: azure/login@v1
      with:
        creds: '${{ secrets.AZURE_CREDENTIALS }}'
    - name: Test azure login by getting details
      run: az account show
```

That should be us good to go! Lets add our changes to git, commit them, and push them back up. Once that's done check on your new workflow result in the actions tab and see what has happened. If you see something like the following, you're all set.



Tasks

Ok, we're all set and connected. Time to make it start doing something useful for us. But first one important note

Billing

For Azure we got $100 free and some specific services free for 12 months. On Github we get some free minutes *per month*. You can find the details here

https://docs.github.com/en/billing/managing-billing-for-github-actions/about-billing-for-github-actions
Free accounts like yours get 2,000 minutes per month which is quite a lot for what we're doing in this practical. But if you're playing with it on your own after this practical keep that limit in mind.

Task 1

First thing we need to do, make it submit an azure ML job. Give that a go.

Tip: You've been doing this command almost every week, just have GitHub do it this time the same way it's running the other commands.

Once you think you've got it commit and push the changes, then GitHub will run it.

Task 2

Running these fully automatically can be great. In a DevOps environment that is absolutely what we would want to do. We also want some of that here in MLOps. Things like our Unit tests and Integration tests (remember we talked about those?) which are going to test the code itself is doing what we want it to. We want those running any time we change something.

But, we might not want the training to run every… single… time…
Training is slow, it takes a lot of resources, and this workflow in practice probably runs the training on our entire dataset. We don't want to do that for every little change. We want to be able to update the code and send it to our team through the repository and let them test it first.

So to mitigate that we want to have a "branch" on git that triggers this workflow. Call it "staging" for example. This is where we're going to send the code only when we think it's ready. So, we only want to trigger the main training job when we push to that branch.

Your task: Update the workflow to only trigger when you push to a specific branch (e.g. staging). Test it works.

Tip: Some details about branching if you're not familiar with it yet
https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-branches

Task 3

This would be a good time to start looking at those unit tests we talked about. Find a python unit testing framework like unittest (or choose your own)
https://docs.python.org/3/library/unittest.html

Your task: Set up a unit test to confirm that the "filter_columns" function in the model.py file returns exactly two columns. Once you've done that set up the workflow to check the unit tests pass before you submit the job for training.

The data set is in pandas, it's documentation will show you how to look at what columns are available.

Tip: There is a lot of available resources for putting unit tests in GitHub actions. Some from GitHub themselves such as https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-python

Task 4

Another option if we're really keen on automatic training on every push; We could have one workflow which runs on most of the branches and only trains on a small subset of our data. Then once we're happy we move it to the "staging" branch which trains it on the bigger dataset.

Your task: Have a go at setting that up with training a small dataset on any branch except staging and the full dataset only on staging

Tip: you'll need to find out how to exclude a branch instead of including one for the push event, and have a second smaller dataset uploaded.

Final thoughts

Ok, let's recap again.

- We've set up automatic training.
- We can run small-scale training tests by pushing our code
- We can run larger ones by moving it into the staging branch when we think it's ready to go
- Our code doesn't get sent to Azure unless the unit tests pass

This is great and has reduced the amount of manual work we need to do. Hopefully you can all see why this is handy and helps us. But there's something missing here from what we've done previously.

Deployment! But deployment is a lot more complicated here.

(P*art of this is because there's actually a missing feature in Azure ML at the minute that makes doing automatic deployment like this with their built in model registration and mlflow awkward or I would have it in here as a required task. Remember that catch about PaaS where we don't have control? This is why that can hurt)*

It's also likely to be case specific to how your organisation handles their deployments and what tools are used. But it's worth having a quick think about how this could work.

Remember, we do **not** want to re-train the model between staging/testing and deployment. That's just asking for trouble. Lots and lots of trouble.

Broadly the steps would go like this. First we test the model as we have done and then add a manual button driven workflow to deploy it to our final quality assurance testing endpoint. That button is going to do the following:

1. Using the CLI download the latest revision of the model
2. Using Docker like we did previously, package the model into a container image
3. Push that image to a container registry
4. Use the updated details to build a new YAML file describing the new deployment
5. Use the CLI to add that new deployment to the endpoint or update the old one

You've done almost all of that already.

- Azure CLI to get information
- Docker to package a model
- Use YAML files to describe Azure ML resources
- Use the CLI with YAML to perform actions/changes/jobs

The main new things for you would be pushing the image to the container registry and building the YAML file in the middle of a job step. That is a much more programmer oriented task which is part of why I'm leaving it out here. You'd be perfectly justified in getting help from the development or operations teams to set that up in a real scenario.

But if you fancy a challenge… Give it a go! It would be good experience for industry.

Shutdown now you're done

Azure will automatically scale down the compute cluster for us which will save money.

BUT IT WILL STILL CHARGE US FOR THE OTHER RESOURCES E.G. STORAGE

Now we're done with the practical we want to shut down the azure resources to stop it spending our credits. The best way to do this is to delete the resource group we created all the resources in (which is why keeping them all together in a resource group is handy). If you didn't do that you will need to delete them individually.

BE SURE TO DELETE THEM BEFORE YOU GO

Go to the resource groups section in azure or search the resource group name you created earlier in the search bar at the top. Once you've selected the resource group you'll see a "Delete resource group" button at the top. After clicking it you will be prompted to type the name of the resource group to ensure that you intend to delete the it. See screenshot below for an example.

Be sure to keep checking your balance frequently to make sure you have not left something running by accident using your credits.