Workshop 3 Lab: CICD Pipelines

Enrique Alejo (enrique.m.alejo@gmail.com)

Digital Transformation Workshops - Universidad Pontificia de Comillas (ICAI)

1 Prerequisites

1. You must have completed lab 1.
2. If you work on a Windows operating system: WSL: Windows Subsystem for Linux - allows running Linux distributions on Windows
3. Docker Desktop: Tool for building and sharing containerized applications
4. Add the following prerequisites to your pyproject.toml file:

```
[tool.poetry.dependencies]
python = ">=3.12,<3.13"
seaborn = "^0.13.2"
pandas = "^2.2.3"
scikit-learn = "^1.6.0"
requests = "^2.32.3"
uvicorn = "^0.34.0"
fastapi = "^0.115.7"
toml = "^0.10.2"
```

2 Objectives

1. Understand how to write unit tests with pytest.

2. Understand how to mock out dependencies with pytest-mock.
3. Create functional APIs with FastAPI.
4. Run containers with Docker.
5. Create CICD pipelines using GitHub Actions to test, build, release and deploy containers to the cloud.

3 Making our code accessible through APIs

3.1 Understanding APIs

Right now our code is only accessible if we download it and run the main.py file. This has several limitations:

1. Users need to have Python installed on their machine
2. They need to download and manage all the dependencies
3. If we release a new software version, they need to update their software as well.
4. Each user needs their own copy of the code and data

To solve these limitations, we can create an API (Application Programming Interface) for our code. An API acts as a bridge that allows different software applications to communicate with each other. By creating an API, the benefits we can gain include:

1. Make our code accessible over the internet.
2. Allow other applications to use our functionality without needing to understand the implementation.
3. Allow other applications to use our functionality without needing to understand the implementation. Just like when you order Pizza at a restaurant, you do not care about the implementation details, you just care about receiving your Pizza.

4. Control access and usage of our application

3.2 Hello World of APIs

We will use FastAPI to expose our application.

Create a file called *lab2.py* in the same directory as *lab1.py* and add the following code:

```python
import uvicorn
from fastapi import FastAPI
from fastapi.responses import FileResponse
from pathlib import Path
import toml
from dtw_lab.lab1 import (
    read_csv_from_google_drive,
    visualize_data,
    calculate_statistic,
    clean_data,
)

# Initialize FastAPI application instance
# This creates our main application object that will handle
   all routing and middleware
app = FastAPI()


# Server deployment configuration function. We specify on
   what port we serve, and what IPs we listen to.
def run_server(port: int = 80, reload: bool = False, host:
   str = "127.0.0.1"):
    uvicorn.run("dtw_lab.lab2:app", port=port, reload=reload,
    host=host)

#Define an entry point to our application.
@app.get("/")
def main_route():
    return {"message": "Hello world"}
```

This is the hello world of APIs. It creates a FastAPI instance called app, defines a function run_server that uses uvicorn to start the web server. Finally, it includes a single route handler for the root path ("/") that responds to GET requests by returning a simple JSON message "Hello world". When run, this creates a minimal web API that you can access through a web browser or API client.

To run this, lets add some scripts to our pyproject.toml file so that we can directly run the server using Poetry.

```
[tool.poetry.scripts]
start-server-dev = "dtw_lab.lab2:run_server(port=8000,reload=
    True)"
start-server = "dtw_lab.lab2:run_server(reload=False,host
    ='0.0.0.0')"
```

Note that we have created two entry points:

1. start-server-dev: This is a development server configuration that runs on port 8000 with hot reloading enabled (reload=True), meaning the server will automatically restart when code changes are detected. This is ideal for local development.
2. start-server: This is a production server configuration that runs with reloading disabled (reload=False) and is set to listen on all network interfaces. This configuration is suitable for production deployment where you want the server to be accessible from external network connections.

These entry points can be executed using

```
poetry run start-server-dev
```

or

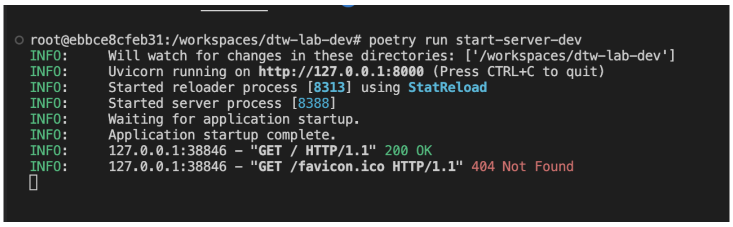
```
poetry run start-server
```

respectively.

Start the dev server as shown in figure 1, and check that you can access the API through your localhost on port 8000 as shown in figure 2

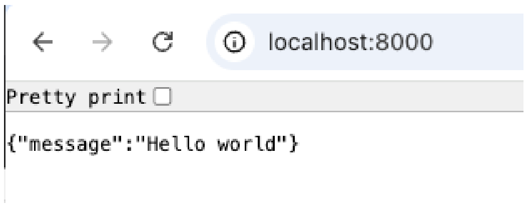
3.3 Exposing our functions through APIs

Now, it is your turn. In the *lab2.py* file, create the following routes and complete them.



```
○ root@ebbce8cfeb31:/workspaces/dtw-lab-dev# poetry run start-server-dev
  INFO:     Will watch for changes in these directories: ['/workspaces/dtw-lab-dev']
  INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
  INFO:     Started reloader process [8313] using StatReload
  INFO:     Started server process [8388]
  INFO:     Waiting for application startup.
  INFO:     Application startup complete.
  INFO:     127.0.0.1:38846 - "GET / HTTP/1.1" 200 OK
  INFO:     127.0.0.1:38846 - "GET /favicon.ico HTTP/1.1" 404 Not Found
```

Figure 1: Starting the server

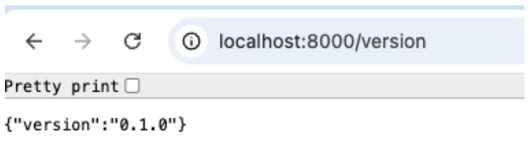


```
{"message":"Hello world"}
```

Figure 2: Accessing the API

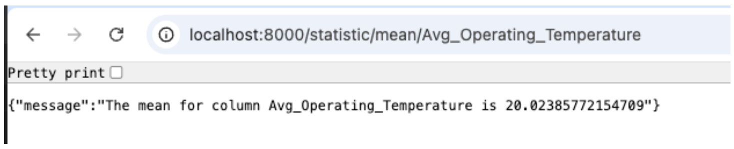
```
@app.get("/statistic/{measure}/{column}")
def get_statistic(measure: str, column: str):
  #Read the CSV data, clean the data, and calculate the
   statistic.

@app.get("/visualize/{graph_type}")
def get_visualization(graph_type: str):
   #Read the CSV data, clean the data, and visualize it.
   #This should create 3 files in the graphs folder.
   #Based on the graph_type input, return the corresponding
   image
   # HINT: Use FileResponse

@app.get("/version")
def get_visualization_version():
    #Using the toml library, get the version field from the "
   pyproject.toml" file and return it.
```

When you implement them and run the dev server again, you can call the APIs from your browser and should see results similar to the ones shown in figures 3, 4 and 5



```
{"version":"0.1.0"}
```

Figure 3: Version API



```
{"message":"The mean for column Avg_Operating_Temperature is 20.02385772154709"}
```

Figure 4: Statistics API

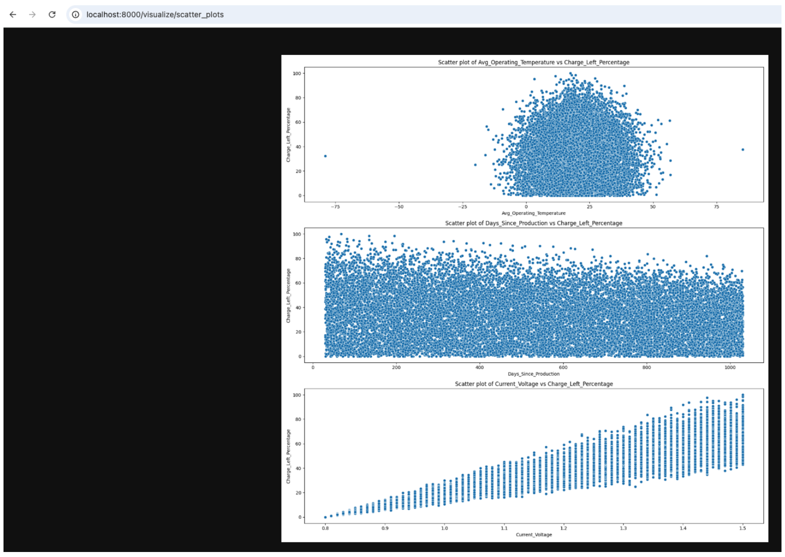

localhost:8000/visualize/scatter_plots
Scatter plot of Avg_Operating_Temperature vs Charge_Left_Percentage
Charge_Left_Percentage
Avg_Operating_Temperature
Scatter plot of Days_Since_Production vs Charge_Left_Percentage
Charge_Left_Percentage
Days_Since_Production
Scatter plot of Current_Voltage vs Charge_Left_Percentage
Charge_Left_Percentage
Current_Voltage

Figure 5: Visualize API

4 Unit tests

In lab 1 we finished with the open ended question:

> How does the approver of the PR know that the new code works as expected?

The answer: tests.

Tests in software development are quality checks that help developers ensure their code works correctly. Think of them as a series of automated experiments that verify if different parts of a program behave as intended. Developers write tests to check if their code produces the expected results. For example, if you have a function that adds two numbers, a test would verify that 2 + 2 actually equals 4 when run through that function. Tests help catch bugs early, make it safer to make changes to the code, and serve as documentation for how the software should work. They can range from simple checks of individual functions (unit tests) to complex scenarios that test entire systems working together (end to end tests). When a developer makes changes to the code, they can quickly run these tests to make sure they haven't accidentally broken anything that was working before.

4.1 Writing our first test

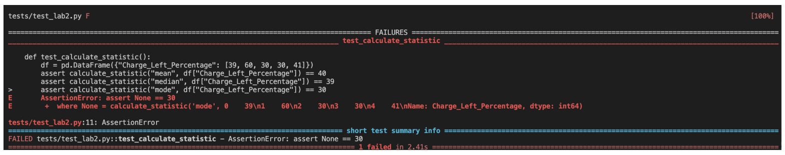
In lab 1, we did not write a single test. When a colleague ran a PR, we would have to clone and manually test the software to check it still works. In this section, we will start writing unit tests to add coverage to our code.

We will start testing the calculate_statistic function. Create a file in the *tests* folder and name it *test_ lab2.py* and add:

```
from src.dtw_lab.lab2 import get_statistic
import pandas as pd
import pytest


def test_calculate_statistic():
    df = pd.DataFrame({"Charge_Left_Percentage": [39, 60, 30,
    30, 41]})
    assert calculate_statistic("mean", df["
Charge_Left_Percentage"]) == 40
    assert calculate_statistic("median", df["
Charge_Left_Percentage"]) == 39
    assert calculate_statistic("mode", df["
Charge_Left_Percentage"]) == 30
```

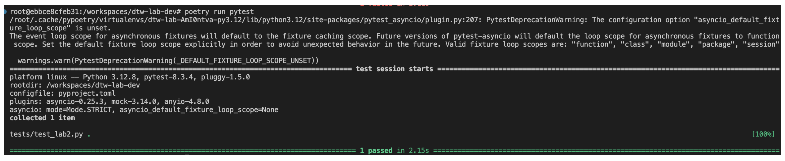
This test uses pytest to verify the calculate_statistic function by creating a test DataFrame with battery charge percentages and asserting that the mean (40), median (39), and mode (30) calculations are correct for the given data. To run the test, you can simply execute pytest in the terminal from your projects root directory - pytest will automatically discover and run any Python files that start with test_ and any functions within those files that start with test_.

Within the poetry shell, execute the *pytest* command. You should see something like in figure 6. It is showing us that one of our assertions is not true. Hence, we have found a bug in our code. Fix it and run pytest again. Now, you should see a passing test as shown in figure 7.



```
tests/test_lab2.py F                                                                                   [100%]

========================================== FAILURES ==========================================
__________________________________ test_calculate_statistic __________________________________

    def test_calculate_statistic():
        df = pd.DataFrame({"Charge_Left_Percentage": [39, 60, 30, 30, 41]})
        assert calculate_statistic("mean", df["Charge_Left_Percentage"]) == 40
        assert calculate_statistic("median", df["Charge_Left_Percentage"]) == 39
>       assert calculate_statistic("mode", df["Charge_Left_Percentage"]) == 30
E       AssertionError: assert None == 30
E        +  where None = calculate_statistic('mode', 0    39\n1    60\n2    30\n3    30\n4    41\nName: Charge_Left_Percentage, dtype: int64)

tests/test_lab2.py:11: AssertionError
================================== short test summary info ===================================
FAILED tests/test_lab2.py::test_calculate_statistic - AssertionError: assert None == 30
====================================== 1 failed in 2.41s =====================================
```

Figure 6: Failing unit test

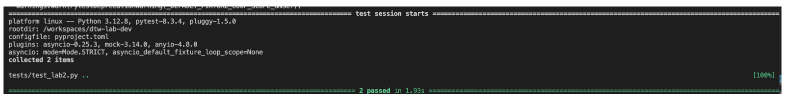


```
root@ebbce8cfeb31:/workspaces/dtw-lab-dev# poetry run pytest
/root/.cache/pypoetry/virtualenvs/dtw-lab-AmI0ntva-py3.12/lib/python3.12/site-packages/pytest_asyncio/plugin.py:207: PytestDeprecationWarning: The configuration option "asyncio_default_fixt
ure_loop_scope" is unset.
The event loop scope for asynchronous fixtures will default to the fixture caching scope. Future versions of pytest-asyncio will default the loop scope for asynchronous fixtures to function
 scope. Set the default fixture loop scope explicitly in order to avoid unexpected behavior in the future. Valid fixture loop scopes are: "function", "class", "module", "package", "session"

  warnings.warn(PytestDeprecationWarning(_DEFAULT_FIXTURE_LOOP_SCOPE_UNSET))
==================================== test session starts =====================================
platform linux -- Python 3.12.8, pytest-8.3.4, pluggy-1.5.0
rootdir: /workspaces/dtw-lab-dev
configfile: pyproject.toml
plugins: asyncio-0.25.3, mock-3.14.0, anyio-4.8.0
asyncio: mode=Mode.STRICT, asyncio_default_fixture_loop_scope=None
collected 1 item

tests/test_lab2.py .                                                                                   [100%]

====================================== 1 passed in 2.15s =====================================
```

Figure 7: Passing unit test

4.2 Implementing unit tests for encode_categorical_vars

Now, write your own test for the function encode_categorical_vars and make sure you have two passing tests as shown in figure 8



```
========================== test session starts ==========================
platform linux -- Python 3.12.8, pytest-8.3.4, pluggy-1.5.0
rootdir: /workspaces/dtw-lab-dev
configfile: pyproject.toml
plugins: asyncio-0.25.3, mock-3.14.0, anyio-4.8.0
asyncio: mode=Mode.STRICT, asyncio_default_fixture_loop_scope=None
collected 2 items

tests/test_lab2.py ..                                              [100%]

========================== 2 passed in 1.93s ==========================
```

Figure 8: Added a second test for encode_categorical_vars

4.3 Understanding Mocks

Notice we create a DataFrame in the first test case, instead of using read_csv_from_google_drive
This follows a key testing best practice: avoiding external dependencies and network calls in unit tests. This approach is better because:

1. Unit tests should be fast and reliable (network calls are slow and can fail)
2. Unit tests should be deterministic (external data could change)
3. Unit tests should work offline
4. We have full control over the test data and can design edge cases
5. We will not negatively impact third party dependencies with our failing code.

When functions use external dependencies or network calls, we need to "mock" or "stub" external dependencies. Mocks are objects that simulate the behavior of real dependencies (like databases, APIs, or complex objects) in unit tests. They allow you to replace real dependencies with fake versions that can return predetermined responses. Take the following example:

```python
def test_api_call(mocker):
    # Setup the mock
    mock_get = mocker.patch('mymodule.requests.get')
    mock_get.return_value.json.return_value = {'data': '
   fake_response'}

    # Run the function
    result = my_function_that_calls_api()

    # Assertions
    assert result == 'fake_response'
    assert mock_get.called
    mock_get.assert_called_once()  # Can use more specific
   assertions
```

This test case uses pytest and pytest-mock to verify a function that makes an API call, but without actually making the network request. It creates a mock (fake) version of the requests.get method that returns a predefined response {'data': 'fake_response'}, then calls the function under test (my_function_that_calls_api()), and finally verifies three things: that the function returns the expected fake response, that the mock API call was actually made, and that it was called exactly once.

4.4 Writing a unit test with mocks

If you implemented the *get_statistic* function correctly in section 3.3, it should be using the *read_csv_from_google_drive* function. Write a test for the *get_statistic* function, and remember to mock out external dependencies.

5 Dockerizing our application

Now, thanks to unit tests, we have working code in our repository. We need to package it so that it is easy to deploy to production in the next steps.

Docker containerization is a powerful option for packaging applications. With Docker, you create a Dockerfile that specifies your application's environment, dependencies, and runtime configuration. This ensures your application runs consistently across different systems and makes deployment much simpler. Later on, we will see how easy it is to deploy our application. Docker containers are self-contained units that include everything needed to run your application, making them excellent for microservices architecture or applications with complex dependencies.

The ability to launch stateless instances of your application helps to improve scalability, consistency and deployment flexibility.

Consider a scenario where your application crashes. With Docker, you can spawn up a new container from a pre-created image and guarantee that it will behave as the previous one.

Now consider that you have to handle thousands of concurrent requests. You can launch more Docker containers to split the load between them. These are just a few of the use cases of containers.

We will now Dockerize our Application. At the root level of your directory, add a Docker file that runs all the necessary steps we have needed to make our application run.

Avoid adding unnecessary dependencies (for example, we won't need to install Docker inside Docker in this case).

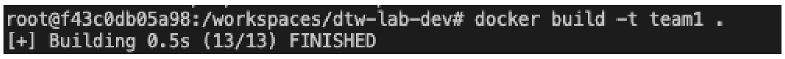
Your Dockerfile should start with

```
FROM python:3.12
```

As an end result, you should be able to execute the following commands:

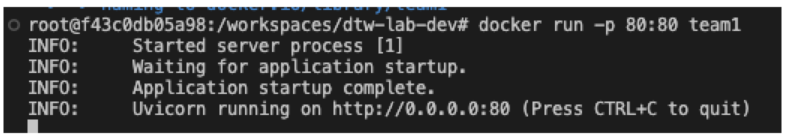
```
docker build -t team${ADD_YOUR_TEAM_NUMBER_HERE} .  #Build your
    container
docker run -p 80:80 team${ADD_YOUR_TEAM_NUMBER_HERE} #Run your
    container and make it accesible through port 80.
```

and see the results shown in figures 9 and 10



```
root@f43c0db05a98:/workspaces/dtw-lab-dev# docker build -t team1 .
[+] Building 0.5s (13/13) FINISHED
```

Figure 9: Build Docker image



```
root@f43c0db05a98:/workspaces/dtw-lab-dev# docker run -p 80:80 team1
INFO:     Started server process [1]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://0.0.0.0:80 (Press CTRL+C to quit)
```

Figure 10: Run the docker image

Tip: to test commands interactively, instead of blindly adding them in the Dockerfile and trusting they will work, you can launch an interactive session to your Docker container by building an image with the layers you know that work and running:

```
docker run -it --rm   --entrypoint  /bin/bash team${
    ADD_YOUR_TEAM_NUMBER_HERE}
```

6 Creating the Pipelines

We have exposed our app through an API, written unit tests and dockerized it. We will now create the CICD pipelines that orchestrate the process to reach production safely.

We will do this using GitHub Actions. In GitHub Actions, you create workflow files using YAML syntax, which are stored in the .github/workflows directory of your repository. These YAML files define the entire automation process, including when the workflow should run, what environment it needs, and the specific steps it should execute. Each workflow file can contain one or more jobs that can run sequentially or in parallel, and each job consists of individual steps that perform specific tasks.

In our case, we will create two workflows. One that gets triggered with pull requests, and one that is triggered on merge requests.

On pull requests, we will have the workflow shown in figure 11.

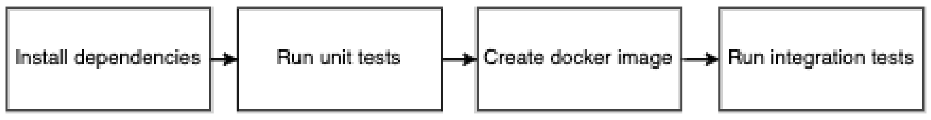


Figure 11: GitHub actions PR Pipeline

It will:

1. Install dependencies: install things like Poetry Python and Docker in the GitHub Actions runner.
2. Run the unit tests that we defined.
3. Build a docker image.
4. Based on the built Docker image: run integration tests. These check that our Docker container works as expected. Instead of going as deep into the code as unit tests, here we treat the Docker image as a black box and check that it responds appropriately.

This will give Pull Request reviewers information on the status of our code, and if it passes tests and builds as expected.

Once a PR is approved and merged, we want to actually deploy our API so that customers can use it. A second pipeline will do this following the steps shown in figure 12.

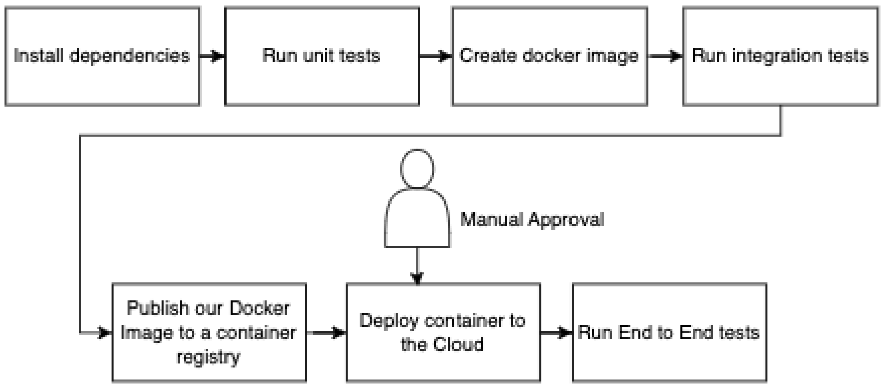


Figure 12: GitHub actions Merge Pipeline

It will:

1. Run the same steps as the PR request: we are just verifying again that our code is stable. More advanced pipelines were timing is critical would probably find a way to avoid this duplication of work.
2. Publish our image: we are storing it in a repository like DockerHub. In the same way that GitHub exists to store our code, Docker registries exist to store our docker images.
3. Deploy to the cloud: we will use the Cloud to deploy our docker image and make it available through the internet.
4. Run end to end tests: now that our application is actually on the cloud, we can check if it is publicly accessible and working as expected for our customers.

Note that this pipeline only deploys once. Typically, we would have different environments, with production being available to customers, staging being a stable copy of our environment, testing used by the quality assurance team and development being used for iterating on the functionality of our code. Check figure 13 for a visual understanding.

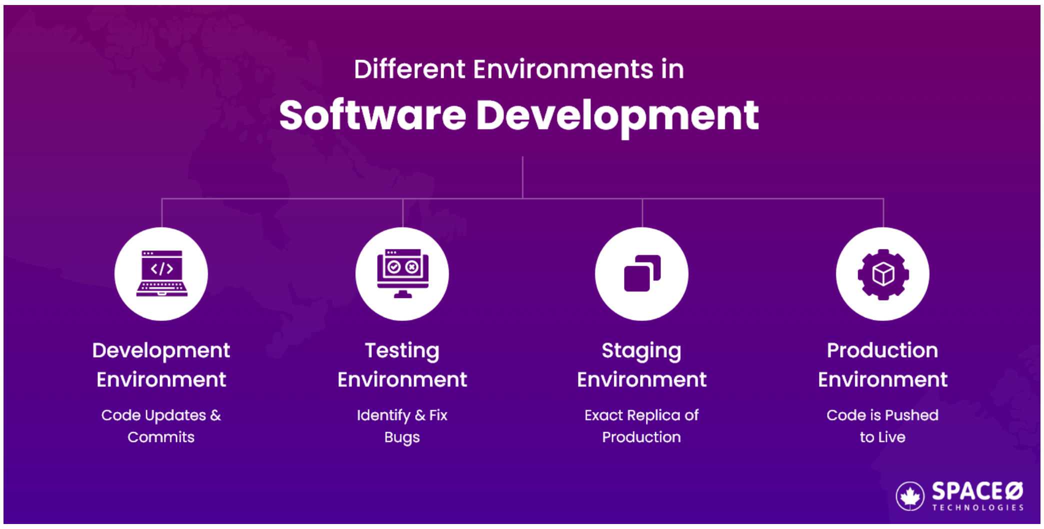


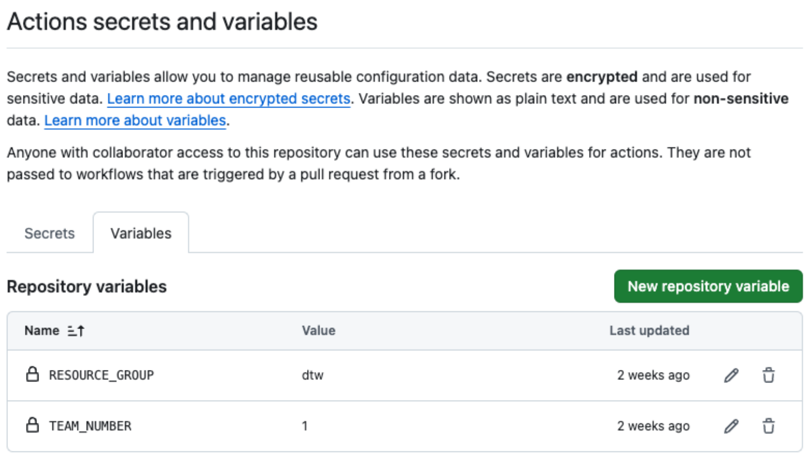
Figure 13: We use different environments to check that our code changes were succesful

6.1 Configuring GitHub with environment variables and secrets

To make our Docker image available in the Cloud or push it to a registry we need to add variables and secrets to our pipelines so that the GitHub runners have the correct credentials.

In your repository settings, go to secrets and variables->Actions. In the variables tab, add the variables shown in figure 14. Ask your professor to grant you a team number and use that.

In the secrets tag, add the secrets shown in figure 15. Ask your professor to provide the actual secret values. As the name states, these are sensitive values. DO NOT SHARE THEM!



Actions secrets and variables

Secrets and variables allow you to manage reusable configuration data. Secrets are **encrypted** and are used for sensitive data. Learn more about encrypted secrets. Variables are shown as plain text and are used for **non-sensitive** data. Learn more about variables.

Anyone with collaborator access to this repository can use these secrets and variables for actions. They are not passed to workflows that are triggered by a pull request from a fork.

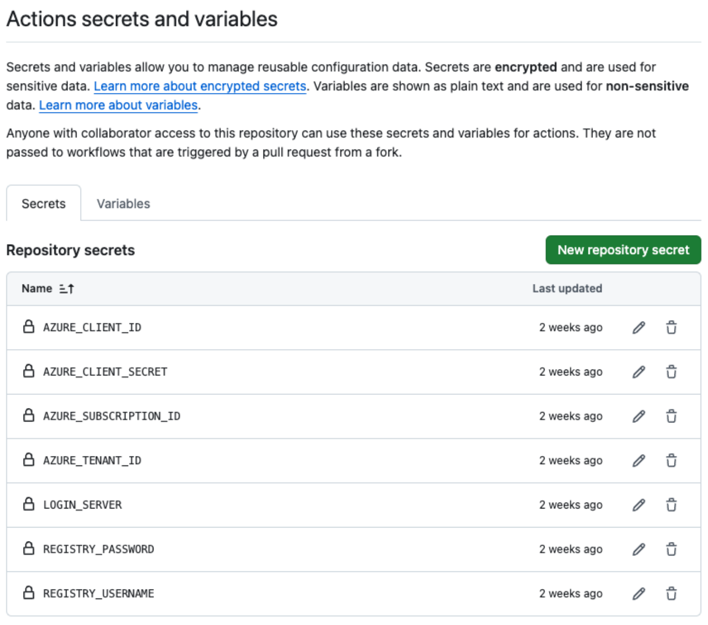
Secrets | Variables

Repository variables

New repository variable

Name	Value	Last updated
RESOURCE_GROUP	dtw	2 weeks ago
TEAM_NUMBER	1	2 weeks ago

Figure 14: Variables used by our pipelines



Actions secrets and variables

Secrets and variables allow you to manage reusable configuration data. Secrets are **encrypted** and are used for sensitive data. Learn more about encrypted secrets. Variables are shown as plain text and are used for **non-sensitive** data. Learn more about variables.

Anyone with collaborator access to this repository can use these secrets and variables for actions. They are not passed to workflows that are triggered by a pull request from a fork.

Secrets | Variables

Repository secrets

New repository secret

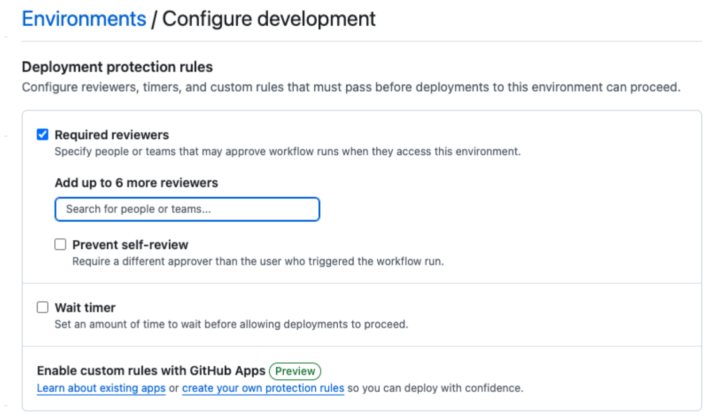
Name	Last updated
AZURE_CLIENT_ID	2 weeks ago
AZURE_CLIENT_SECRET	2 weeks ago
AZURE_SUBSCRIPTION_ID	2 weeks ago
AZURE_TENANT_ID	2 weeks ago
LOGIN_SERVER	2 weeks ago
REGISTRY_PASSWORD	2 weeks ago
REGISTRY_USERNAME	2 weeks ago

Figure 15: Secrets used by our pipelines

6.2 Adding manual approvals

Now that our pipelines have secrets configured, lets add a manual step before deployment occurs, so we do not have unwanted/accidental deployments.

Go to settings->Environments and create an environment named development. Then, check the box "Require reviewers" and grant the appropriate users permissions to approve the deployment action as shown in figure 16.


Environments / Configure development

Deployment protection rules
Configure reviewers, timers, and custom rules that must pass before deployments to this environment can proceed.

☑ Required reviewers
Specify people or teams that may approve workflow runs when they access this environment.

Add up to 6 more reviewers
Search for people or teams...

☐ Prevent self-review
Require a different approver than the user who triggered the workflow run.

☐ Wait timer
Set an amount of time to wait before allowing deployments to proceed.

Enable custom rules with GitHub Apps Preview
Learn about existing apps or create your own protection rules so you can deploy with confidence.

Figure 16: Protecting our environments through manual approval processes.

6.3 Creating the pipeline than runs on PR

Now we will implement two pipelines using GitHub Actions. GitHub looks for files in the folder '.github/workflows' to create pipelines. In Moodle you will find a starting point for the pipelines.

Also from Moodle, get and add the folder 'scripts' and its content to the repository. This includes some helper scripts that will make the process of building, testing releasing and deploying simpler.

1. build.sh: This script is responsible for building a Docker image. It extracts the package version from the pyproject.toml file and builds a Docker image with the tag $LOGIN_SERVER/team$TEAM_NUMBER:$PACKAGE_VE to make the release easier. It also creates a latest tag for the same image.

2. integration_test.sh: This script performs integration testing on the built container. It extracts the package version from pyproject.toml to figure out the tag of the container that it can run. It runs the container locally on port 80 and waits for 20 seconds to allow the container to start. Finally, it sends a curl request to check if the container is running. This is a very simple test, but at least we know our container is able to spin up. Finally, it stops and removes the container.

3. release.sh: this script is responsible for releasing the built container. It extracts the package version from pyproject.toml and logs into the container registry using the Secrets we added to our repository. It pushes the built image to the registry and makes the package version available to the next step in a GitHub Actions workflow by writing it to the $GITHUB_OUTPUT file.

4. deploy.sh: This script handles the deployment of the container to Azure cloud. It logs into Azure using service principal credentials and creates a container in Azure Container Instances. This is the power of the cloud. With one single command, the container you created is available for anyone to consume. In fact, you should already see a running container in http://team{TEAM-NUMBER}.northeurope.azurecontainer.io/. Go ahead and check that you can access it.

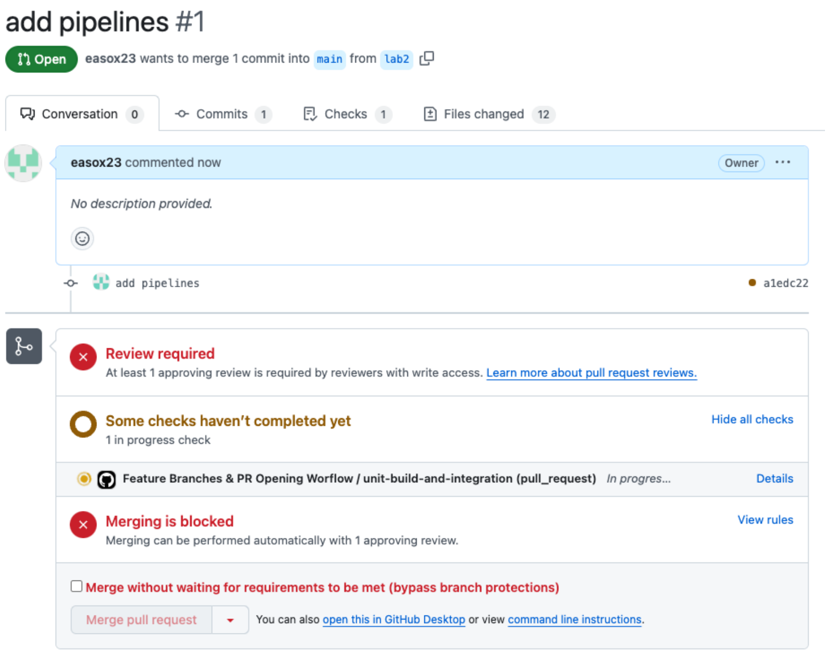
Add all these files to your repository and execute the following command inside the scripts folder to allow them to be used as executables:

```
chmod +x *
```

Create a feature branch for the changes, push, and open a PR. You do not need to modify the content of the files yet.

If we added the files correctly, the pipeline defined in the file pr-openning.yaml will be triggered as shown in figure 17. These checks show up with the PR process to give more information to our code reviewer on the status of our code changes.

Inside the actions section of your repository, you can see what the actual pipeline is running. Figure 18 shows the steps.



add pipelines #1

Open easox23 wants to merge 1 commit into `main` from `lab2`

Conversation 0 | Commits 1 | Checks 1 | Files changed 12

easox23 commented now — Owner

No description provided.

`add pipelines` — a1edc22

Review required
At least 1 approving review is required by reviewers with write access. Learn more about pull request reviews.

Some checks haven't completed yet — Hide all checks
1 in progress check

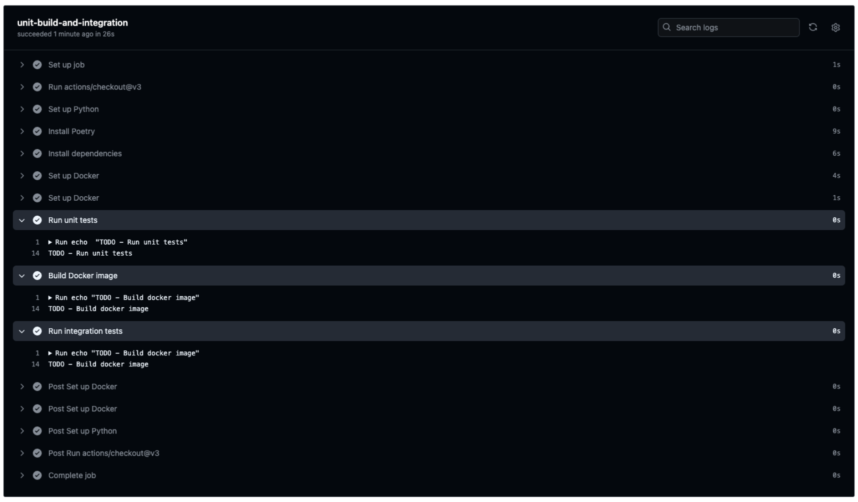
Feature Branches & PR Opening Worflow / unit-build-and-integration (pull_request) *In progres...* — Details

Merging is blocked — View rules
Merging can be performed automatically with 1 approving review.

☐ **Merge without waiting for requirements to be met (bypass branch protections)**

Merge pull request — You can also open this in GitHub Desktop or view command line instructions.

Figure 17: PR running the automatic checks



```
unit-build-and-integration
succeeded 1 minute ago in 26s

> Set up job                              1s
> Run actions/checkout@v3                 0s
> Set up Python                           0s
> Install Poetry                          9s
> Install dependencies                    6s
> Set up Docker                           4s
> Set up Docker                           1s
v Run unit tests                          0s
  ▶ Run echo  "TODO - Run unit tests"
  TODO - Run unit tests
v Build Docker image                      0s
  ▶ Run echo "TODO - Build docker image"
  TODO - Build docker image
v Run integration tests                   0s
  ▶ Run echo "TODO - Build docker image"
  TODO - Build docker image
> Post Set up Docker                      0s
> Post Set up Docker                      0s
> Post Set up Python                      0s
> Post Run actions/checkout@v3            0s
> Complete job                            0s
```

Figure 18: PR Pipeline Steps

6.4 Adding functionality to the pipeline

Figure 18 shows the steps we described in section 6 . We install dependencies (first 7 steps), we run unit tests, we build the docker image, and we run integration tests. Note however, that the tests and Docker steps are not yet implemented. You should now work to add them to the pipeline. Use the files in the scripts folder to help you.

Tip: you can run .sh files as a step in the pipeline. This simplifies debugging. You should not try to add all the bash code to the yaml file.

6.5 Creating the pipeline than runs on Merge to Main

Now we want to build the pipeline that runs once we merge to main. As described in section 6, it will repeat the steps from the PR pipeline, but add release, deployment and end to end test steps.

This pipeline should have 3 jobs.

1. release: Identical to previous pipeline with the addition of running release.sh as the last step. Release should take the Docker image we build and push it to a registry.
2. deploy-to-dev: This job should wait until release step is done. It will take the container image we just released and tell the cloud to run it. You can read more about the cloud service that will run it here. Make sure to tag it as a development environment so that the manual approval rule we defined before takes effect. Also, make sure to add the environment variables that the deploy.sh script needs.
3. e2e-tests: This is not yet implemented in the scripts file. With end to end tests, we want to test the real system that our users will see. You will need to write a script to see that your container is working. It should be accessible at http://team{TEAM-NUMBER}.northeurope.azurecontainer.io/

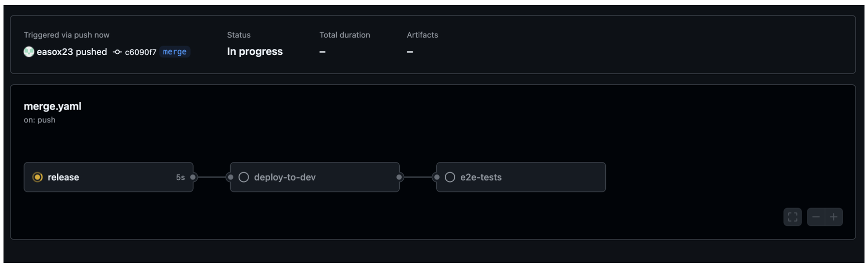
Note that each job runs on a brand new clean container. So for each job you will need to reinstall the needed dependencies. Also note what environment variables the helper scripts need when you are defining your Github actions pipelines. Check the images from section 6.6 to guide you.

Tip: Change the trigger conditions for this pipeline to run on every commit so that you can test that it works without having to merge to main. When you are ready, change the condition back to only pushes to main.

6.6 Deployment results

When your merge pipeline is working, any new changes that reach the main branch should automatically be made available in your public URL. The following figures show what you should observe.

Figure 19 shows the jobs of your pipeline, figure 20 shows the manual approval before deployment. Finally, if you deployed correctly, you should be able to access your container in a public link as shown in figure 21.



Triggered via push now
easox23 pushed c6090f7 merge

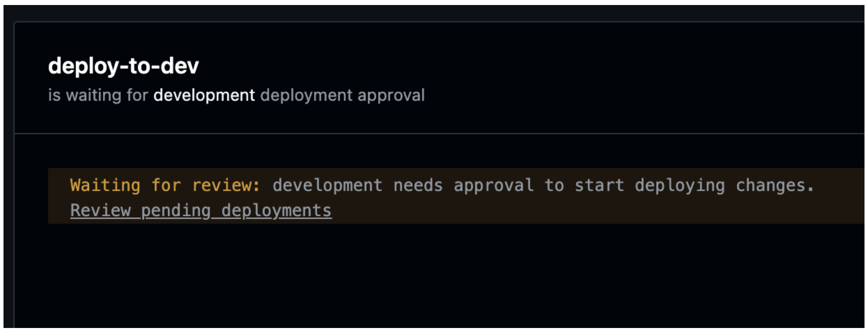
Status
In progress

Total duration
–

Artifacts
–

merge.yaml
on: push

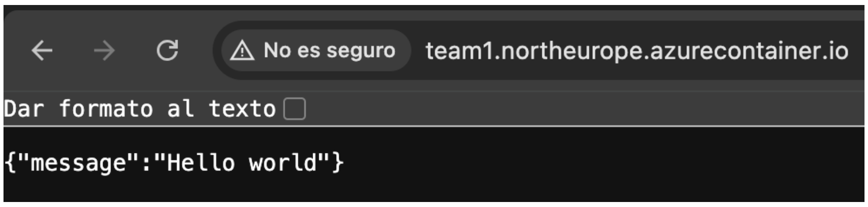
Figure 19: Merge pipeline shown on GitHub



deploy-to-dev
is waiting for **development** deployment approval

```
Waiting for review: development needs approval to start deploying changes.
Review pending deployments
```

Figure 20: Manual step before deployment



```
{"message":"Hello world"}
```

Figure 21: Public access to my container

7 Conclusion

In this lab, we learned about APIs, testing, dockerization and CICD pipelines. We joined these components together to allow us to make our APIs available in the internet. This acts as an introduction to understand how to deliver our software products to customers using DevOps methodologies.

As a deliverable, prepare a single document with your team explaining:

1. Your implementation of the APIs
2. The unit tests you implemented
3. The DockerFile you created
4. PR Pipeline configuration file
5. Merge pipeline configuration file
6. Explanation of your end to end tests
7. Image of your passing pipelines on GitHub

Make sure to add your team number, the student number of all participants and a link to the GitHub repo in the report.