CI/CD pipeline for flask deployment on ECS

Continuous integration (CI)

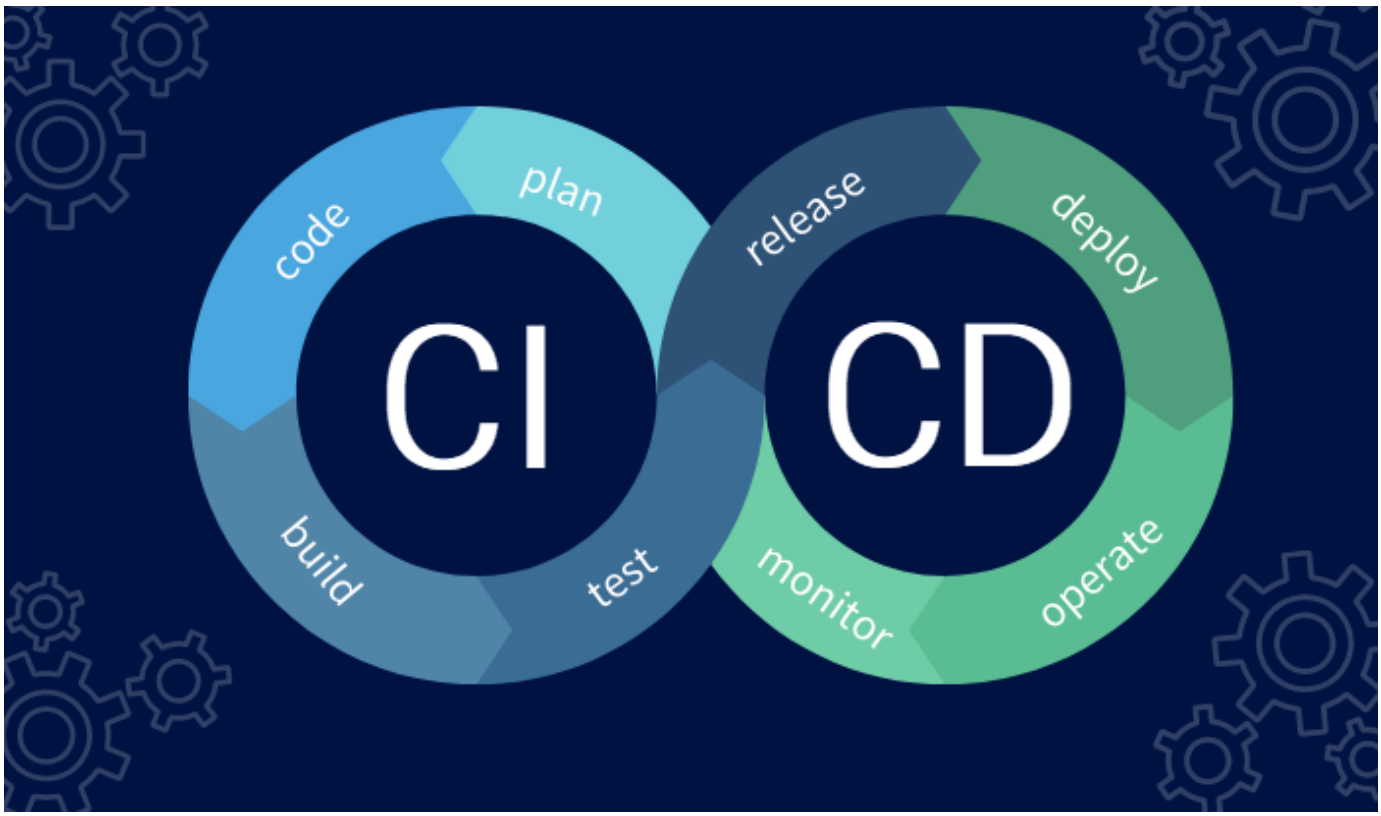
Continuous integration is the practice of integrating all your code changes into the main branch of a shared source code repository early and often, automatically testing each change when you commit or merge them, and automatically kicking off a build. With continuous integration, errors and security issues can be identified and fixed more easily, and much earlier in the software development lifecycle.

continuous delivery/continuous deployment (CD)

Continuous delivery is the automated delivery of completed code to environments like testing and development. CD provides an automated and consistent way for code to be delivered to these environments.

CD is the next step of CI. Every change that passes the automated tests is automatically placed in production, resulting in many production deployments.

> CI/CD allows organizations to ship software quickly and efficiently. CI/CD facilitates an effective process for getting products to market faster than ever before, continuously delivering code into production, and ensuring an ongoing flow of new features and bug fixes via the most efficient delivery method.



Flask app

- we will start with the flask app that we created for the loan prediction model.
  - we already have the flask app ready and running
  - we also have the docker container ready to be deployed
  - we understand the concepts of webapps, deployments and cloud infra

    > the content of the main app.py file is pasted below for reference

  - preferably use a virtual enviornment for this

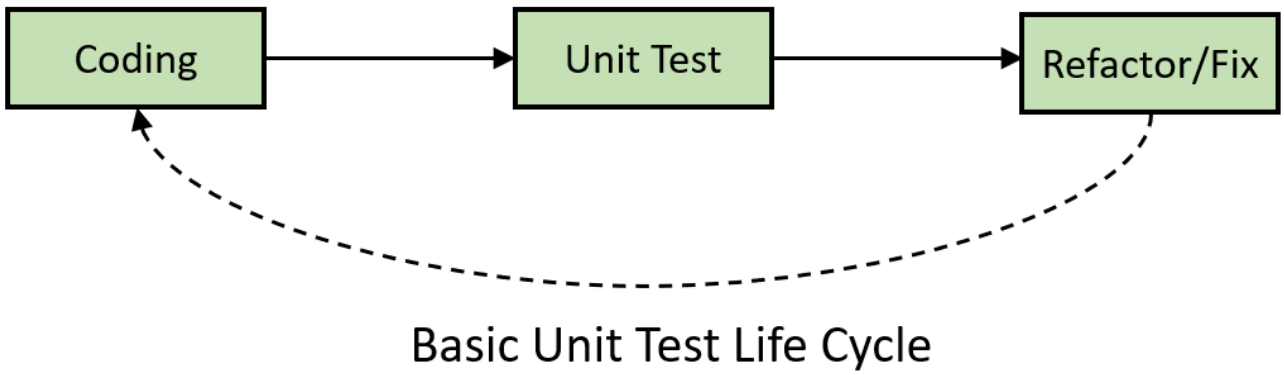
```
import pickle
import flask

from flask import Flask, request, Response, jsonify



## loading the model
model_pickle = open("./artefacts/classifier.pkl", 'rb')
clf = pickle.load(model_pickle)

app = Flask(__name__)



# defining the function which will make the prediction using the data which the user input
@app.route('/predict', methods = ['POST'])
def prediction():
    # Pre-processing user input
    loan_req = request.get_json()
    print(loan_req)

    if loan_req['Gender'] == "Male":
        Gender = 0
    else:
        Gender = 1

    if loan_req['Married'] == "Unmarried":
        Married = 0
    else:
        Married = 1

    if loan_req['Credit_History'] == "Unclear Debts":
        Credit_History = 0
    else:
        Credit_History = 1

    ApplicantIncome = loan_req['ApplicantIncome']
    LoanAmount = loan_req['LoanAmount'] / 1000
```

```
    # Making predictions
    prediction = clf.predict(
        [[Gender, Married, ApplicantIncome, LoanAmount, Credit_History]])

    if prediction == 0:
        pred = 'Rejected'
    else:
        pred = 'Approved'

    result = {
        'loan_approval_status': pred
    }

    return jsonify(result)


@app.route('/ping', methods=['GET'])
def ping():
    return "Pinging Model!!"
```

Unit tests

"Unit tests are typically automated tests written and run by software developers to ensure that a section of an application (known as the "unit") meets its design and behaves as intended"

> Essentially, a unit test is a method that instantiates a small portion of our application and verifies its behavior independently from other parts. A typical unit test contains 3 phases: First, it initializes a small piece of an application it wants to test (also known as the system under test, or SUT), then it applies some stimulus to the system under test (usually by calling a method on it), and finally, it observes the resulting behavior. If the observed behavior is consistent with the expectations, the unit test passes, otherwise, it fails, indicating that there is a problem somewhere in the system under test.



Basic Unit Test Life Cycle

Pytest

Pytest is possibly the most widely used Python testing framework around - this means it has a large community to support you whenever you get stuck. It's an open-source framework that enables developers to write simple, compact test suites while supporting unit testing, functional testing, and API testing.

To install the latest version of pytest, execute the following command −

pip install pytest

```
pip install pytest
```

Confirm the installation using the following command to display the help section of pytest.

```
pytest -h
```

to demonstrate the ease of pytest we ill create a simple python file and run test on it

create a file *square.py* and copy the following conent

- the logic is just a function that returns square of the input number

```
def square(a):
    return a*a
```

create another file *test_square.py* and paste the following content

```
from square import square_num

def test_square_num():
```

```
a=3
res=square_num(a)
assert res==9
```

Pytest will execute all the python files that have the name test_ prepended or _test appended to the name of the script. To be more specific, pytest follows the following conventions for test discovery :

- Given no arguments are specified, pytest collection would begin in testpaths if they are configured: testpaths are a list of directories pytest will search when no specific directories, files, or test ids are provided.
- Pytest would then recurse into directories unless you've told it not to by setting norecursedirs; It's searching for files that begin with test_*.py or end in *_test.py
- In those files, pytest would collect test items in the following order: Prefixed test functions or methods outside of class Prefixed test functions or methods inside Test prefixed test classes that do not have an **init** method.
- We have not specified any arguments, but we have created another script in the same directory called test_square.py: thus, when the directories are recursed the test will be discovered. In this script, we have a single test, test_square_num, to validate our function is working accordingly.
- Pytest requires the test function names to start with test. Function names which are not of format test* are not considered as test functions by pytest. We cannot explicitly make pytest consider any function not starting with test as a test function.

> Note: You may decide to put your tests into an extra directory outside of your application which is a good idea if you have several functional tests or you want to keep testing code and application code separate for some other reason.

Running simple tests

```
(flask_ecs) abdulahad_scaler@Abduls-Air Flask_ecs % pytest test_square.py
============================= test session starts ==============================
platform darwin -- Python 3.9.13, pytest-7.2.0, pluggy-1.0.0
rootdir: /Users/abdulahad_scaler/python/Flask_ecs
plugins: typeguard-2.13.3
collected 1 item

test_square.py .                                                         [100%]

============================== 1 passed in 0.01s ===============================
(flask_ecs) abdulahad_scaler@Abduls-Air Flask_ecs %
```

The output then indicates the status of each test using a syntax similar to unittest:

- A dot (.) means that the test passed.
- An F means that the test has failed.
- An E means that the test raised an unexpected exception.

The special characters are shown next to the name with the overall progress of the test suite shown on the right.

- For tests that fail, the report gives a detailed breakdown of the failure.

Creating tests for our flask app

we have two functions in our flask app

- one that we can ping to see if the app is running
- one that takes in the agruments and predicts if the loan should be approved or not

So we will be creating simple tests for these two functions

```
import pytest
from app import app
import json

@pytest.fixture
def client():
    return app.test_client()


def test_home(client):
    resp = client.get('/ping')
    assert resp.status_code == 200

def test_predict(client):
    test_data={'Gender':"Male", 'Married':"Unmarried",'Credit_History' : "Unclear Debts",'
    resp=client.post('/predict', json=test_data)
    assert resp.status_code == 200
    assert resp.json=={'loan_approval_status': 'Rejected'}
```

Pytest Fixtures

Pytest's invaluable fixtures feature permits developers to feed data into the tests. They are essentially functions that run before each test function to manage the state of our tests. For example, say we have several tests that all make use of the same data then we can use a fixture to pull the repeated data using a single function.

- In Pytest, we can use the fixture function as an input parameter of the test function, and that input parameter is already the return object.

- We have to indicate that the function is a fixture with @pytest.fixture. These specific Python decorations let us know that the next method is a pytest fixture.

test_client

The test client makes requests to the application without running a live server. In order to create the proper environment for testing, Flask provides a test_client helper. This creates a test version of our Flask application, which we used to make a GET and POST call to the app.

> we will pass that client to other functions and call them with the required methods and assert the result
>
> - first test method will ping the ping method and check the response code. 200 means that everything is fine
> - we will create a dummy input pass it to the app and check the result. for the dummy input we create we know that the result should be rejected, so that is what we check

you can run them in you local system and the result should be positive

Github actions

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. You can create workflows that build and test every pull request to your repository, or deploy merged pull requests to production.

GitHub Actions goes beyond just DevOps and lets you run workflows when other events happen in your repository. For example, you can run a workflow to automatically add the appropriate labels whenever someone creates a new issue in your repository.

GitHub provides Linux, Windows, and macOS virtual machines to run your workflows, or you can host your own self-hosted runners in your own data center or cloud infrastructure.

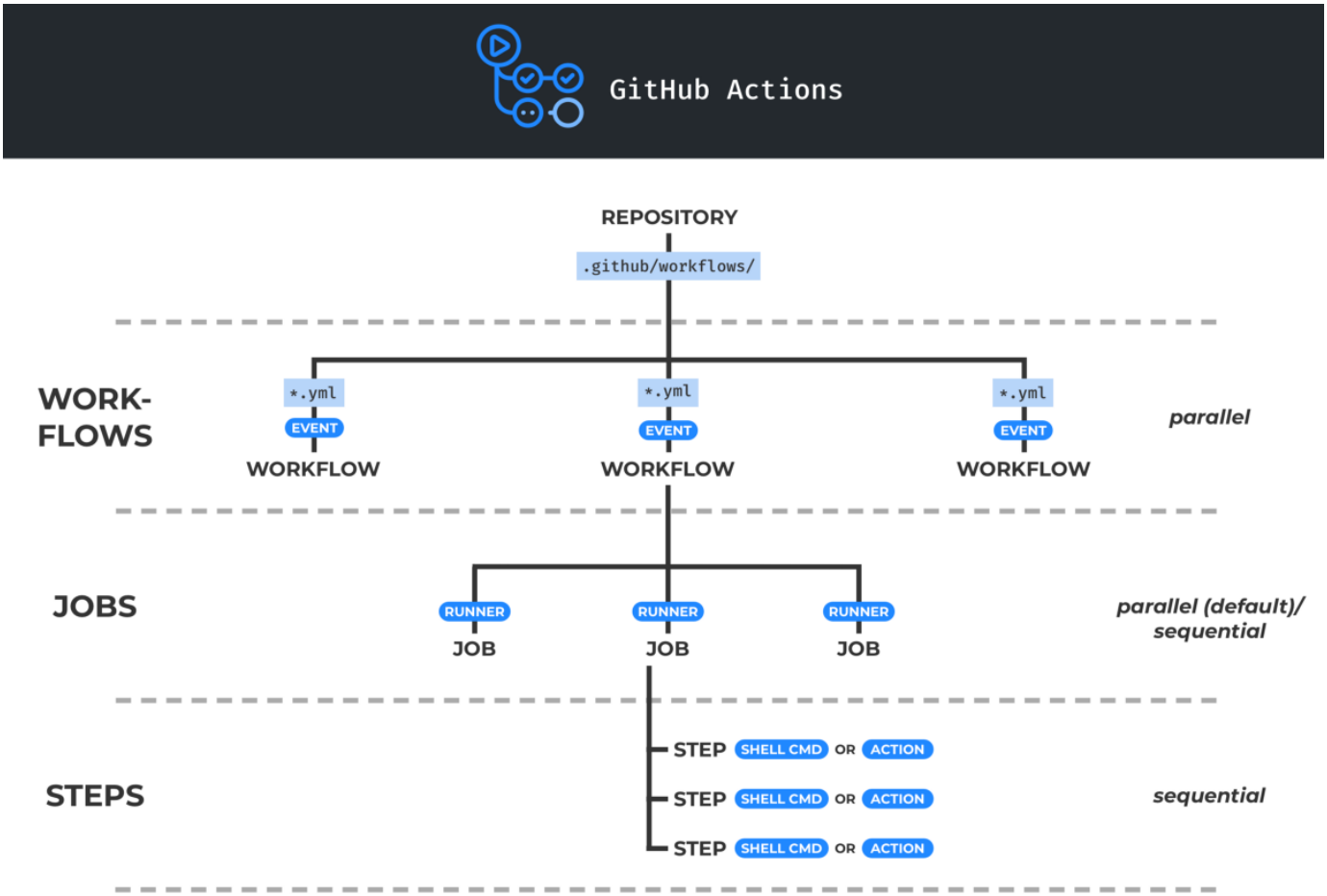
The components of GitHub Actions

You can configure a GitHub Actions workflow to be triggered when an event occurs in your repository, such as a pull request being opened or an issue being created. Your workflow contains one or more jobs which can run in sequential order or in parallel. Each job will run inside its own virtual machine runner, or inside a container, and has one or more steps that either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.

- **Workflows** A workflow is a configurable automated process that will run one or more jobs. Workflows are defined by a YAML file checked in to your repository and will run when triggered by an event in your repository, or they can be triggered manually, or at a defined schedule.Workflows are defined in the .github/workflows directory in a repository, and a

repository can have multiple workflows, each of which can perform a different set of tasks. For example, you can have one workflow to build and test pull requests, another workflow to deploy your application every time a release is created, and still another workflow that adds a label every time someone opens a new issue.

- **Events** An event is a specific activity in a repository that triggers a workflow run. For example, activity can originate from GitHub when someone creates a pull request, opens an issue, or pushes a commit to a repository. You can also trigger a workflow run on a schedule, by posting to a REST API, or manually.
- **Jobs** A job is a set of steps in a workflow that execute on the same runner. Each step is either a shell script that will be executed, or an action that will be run. Steps are executed in order and are dependent on each other. Since each step is executed on the same runner, you can share data from one step to another. For example, you can have a step that builds your application followed by a step that tests the application that was built.



GitHub Actions uses YAML syntax to define the workflow. Each workflow is stored as a separate YAML file in your code repository, in a directory named .github/workflows.

> **There were a lot of commands and syntax to put here so open the below links for github documnetations to understand better**

https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions#about-yaml-syntax-for-workflows

create github actions

```
name: Run Python Tests
on:
  push:
    branches:
      - main


jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Install Python 3
        uses: actions/setup-python@v1
        with:
          python-version: 3.6
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install pytest
          pip install -r requirements.txt
      - name: Run tests with pytest
        run: pytest
```

lets understand this:

- on push branches main we want to perform following jobs. ie: whenever anything is pushed to the main branch following jobs need to be run. here we can specify specific branches of the repository or other activities like pull or fork
- then we define our jobs that needs to be performed
    - Each job runs in a runner environment specified by runs-on, which in our case we want it to be ubuntu-latest
- next we define the steps to be taken
    - actions/checkout: is an official GitHub Action used to check-out a repository so a workflow can access it.
    - define the python evn

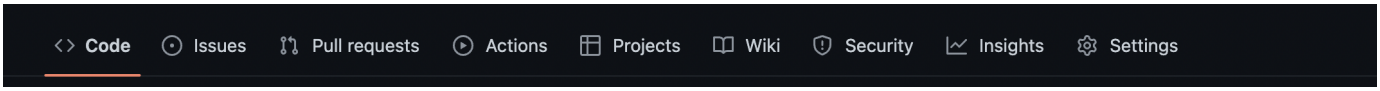
    - run the following commands is basically running those commands and checking their status
    - we preaper the enviornment by running the pip install requirements file and also pytest
    - finally we run pytest to get the results

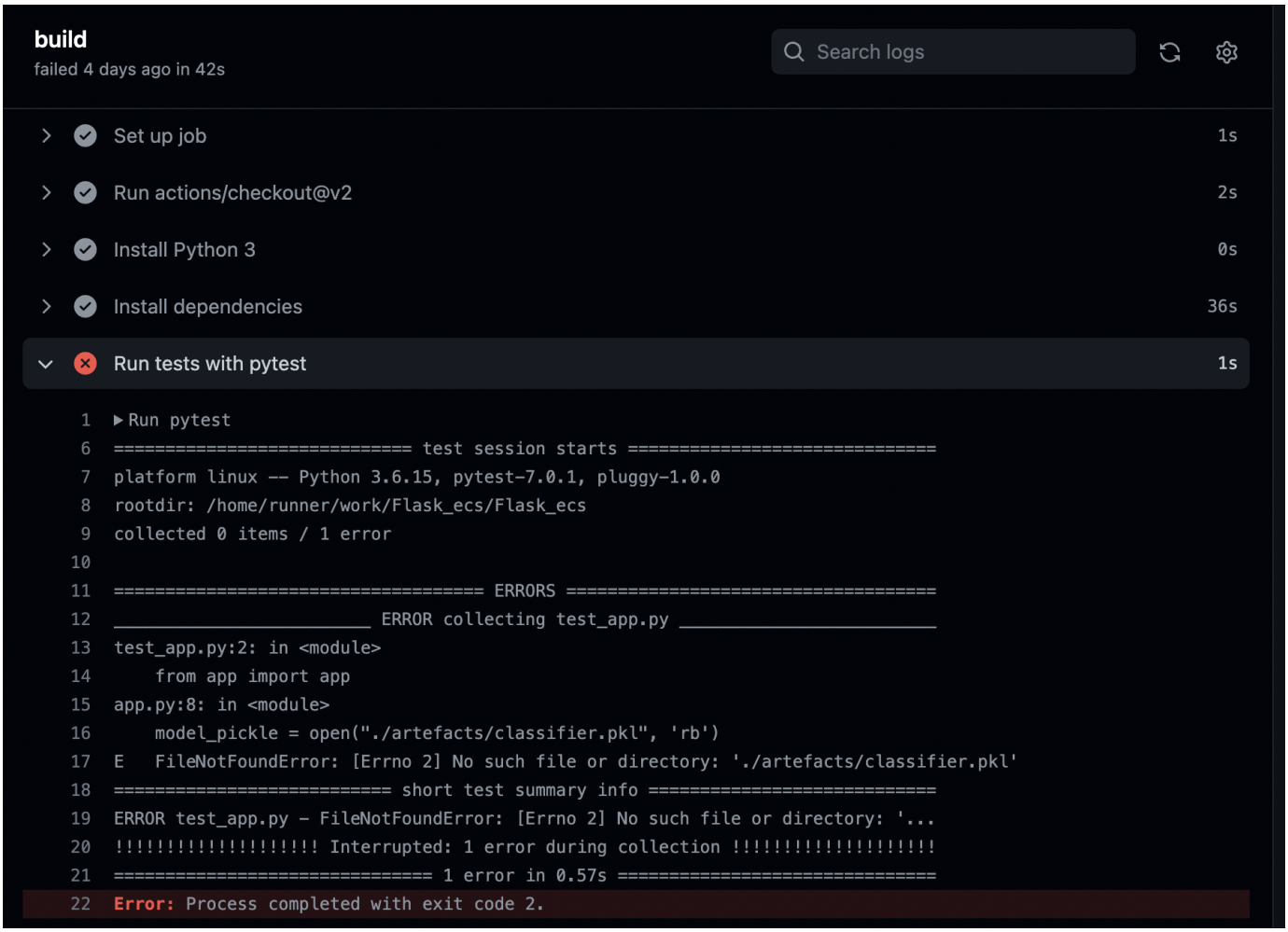
push the following files to the repo

- app.py
- test_app.py
- requirements.txt

as soon as the push is complete and you have the yml file in the correct location setup

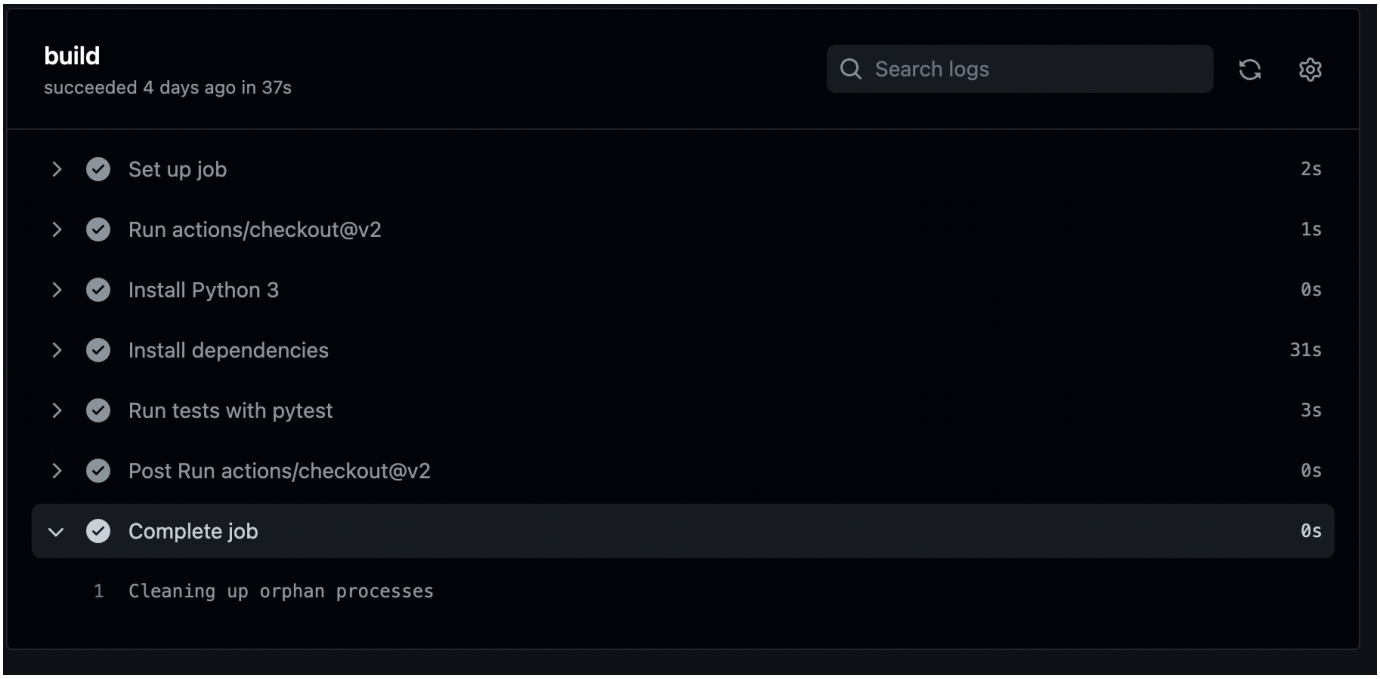
- head to the actions tab in the repo, you will see that it is running a workflow
    - wait for it to complete





- the workflow shows red cross sign that means it has failed. click on it to inspect the reason

- you will see that the there is an error in the run tests with pytest job
- on furthur inspection it shows that there is a file not found error. ie: because we did not upload the model file along with the app, but fortunately our test caught this error and we can rectify it.
- upload the arefact folder as well to the repo
- as soon as the push is complete it will run the workflow again



this time the worklow was successfull