CI/CD pipeline for flask deployment on ECS

Continuous integration (CI)

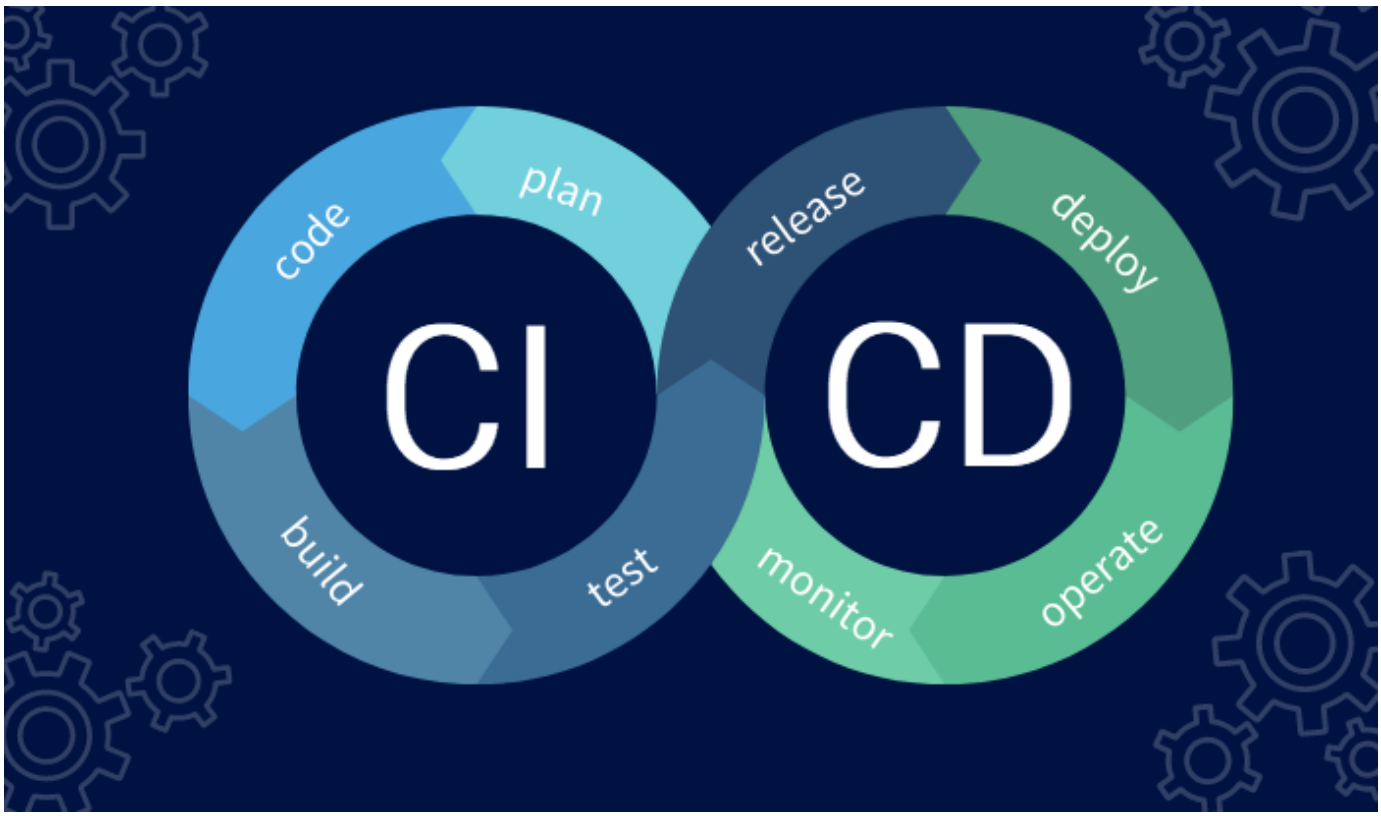
Continuous integration is the practice of integrating all your code changes into the main branch of a shared source code repository early and often, automatically testing each change when you commit or merge them, and automatically kicking off a build. With continuous integration, errors and security 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

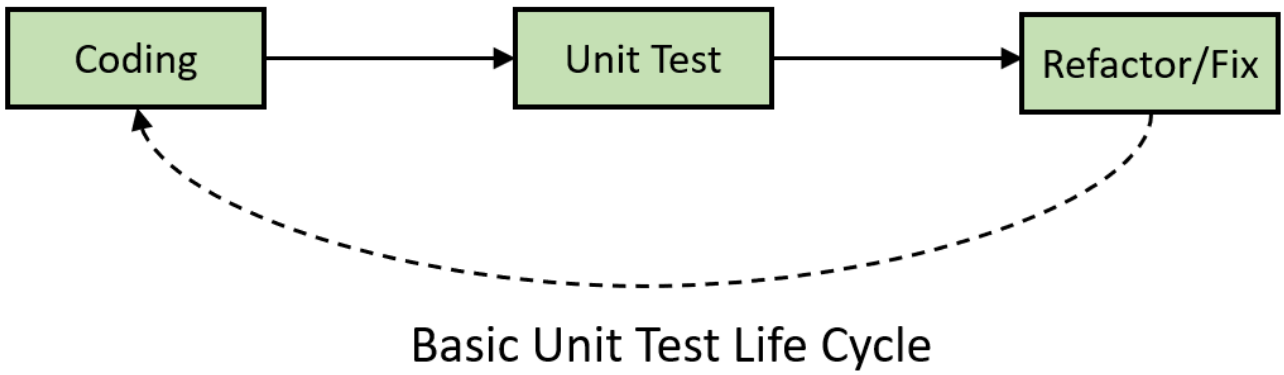
```
import pickle
import flask

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.

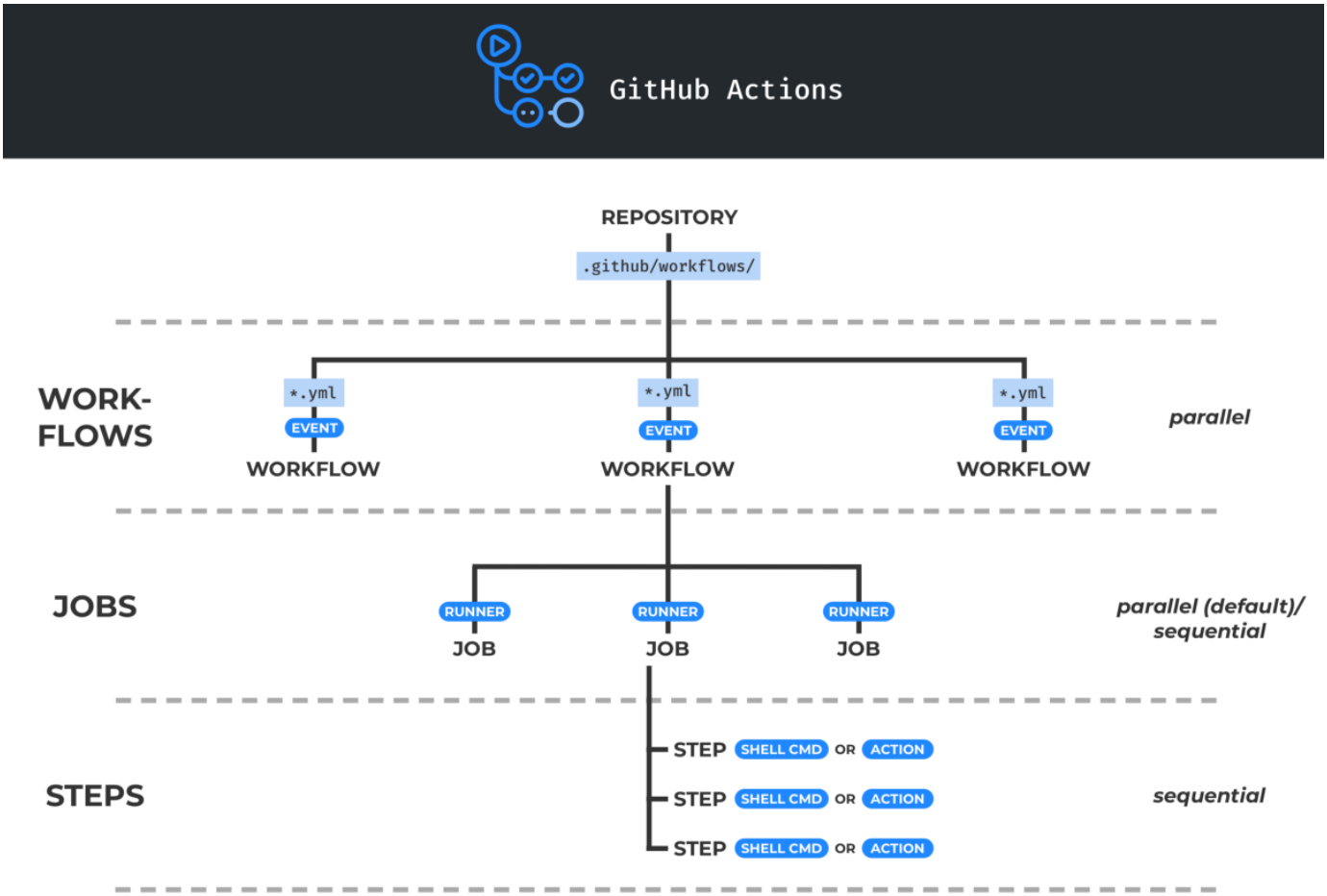
The components of GitHub Actions

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

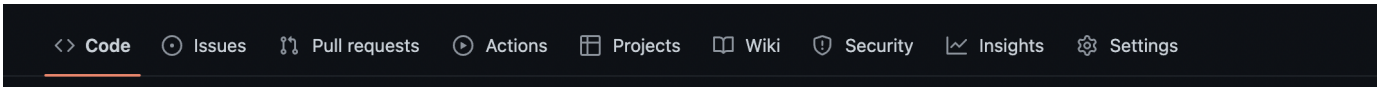
    - run the following commands is basically running those commands and checking their status
    - we preaper the enviornment by running the pip install requirements file and also pytest
    - finally we run pytest to get the results

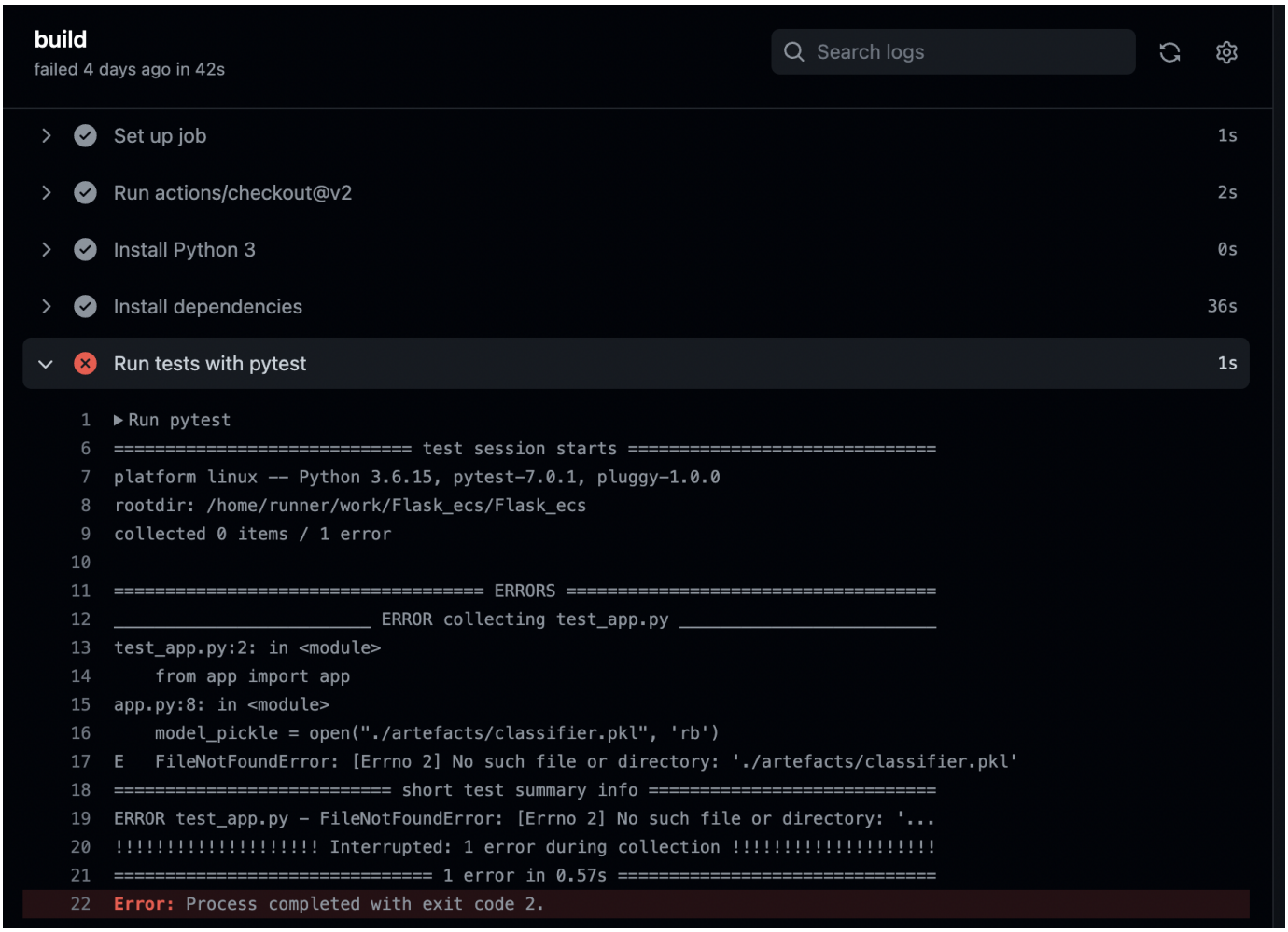
push the following files to the repo

- 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

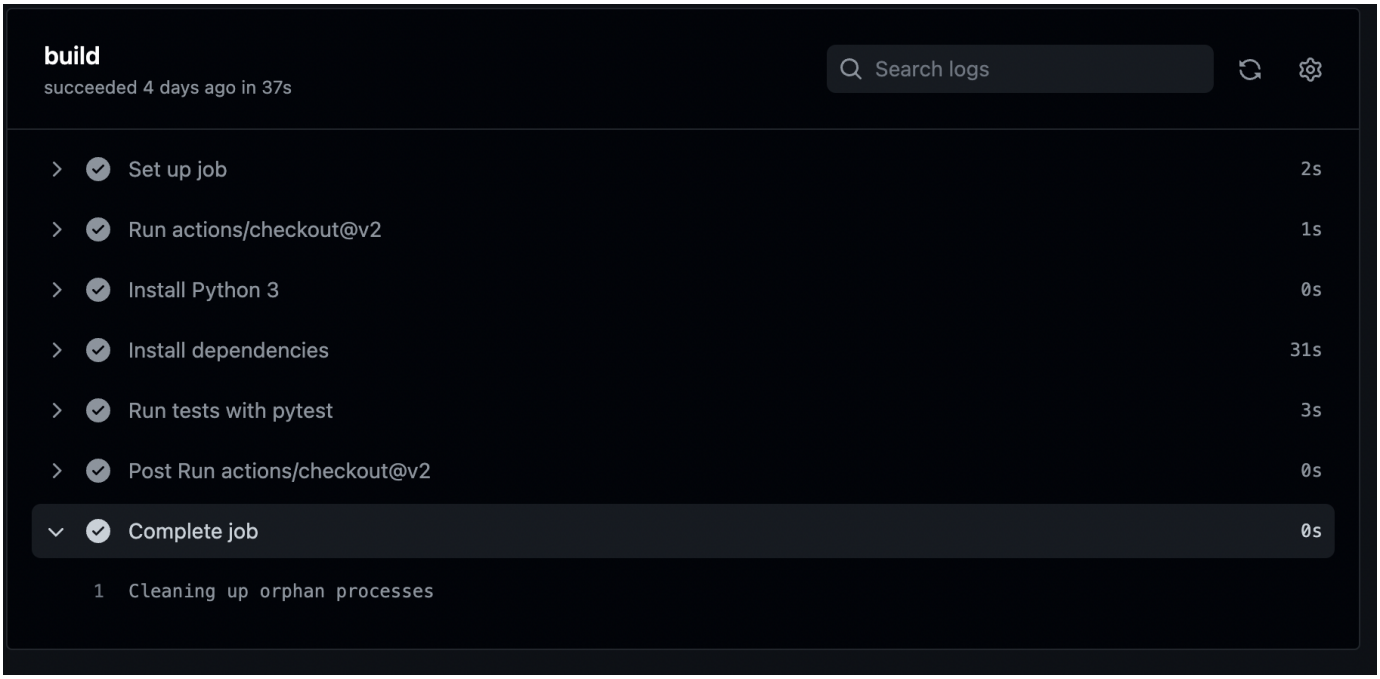
- head to the actions tab in the repo, you will see that it is running a workflow
    - wait for it to complete





- 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



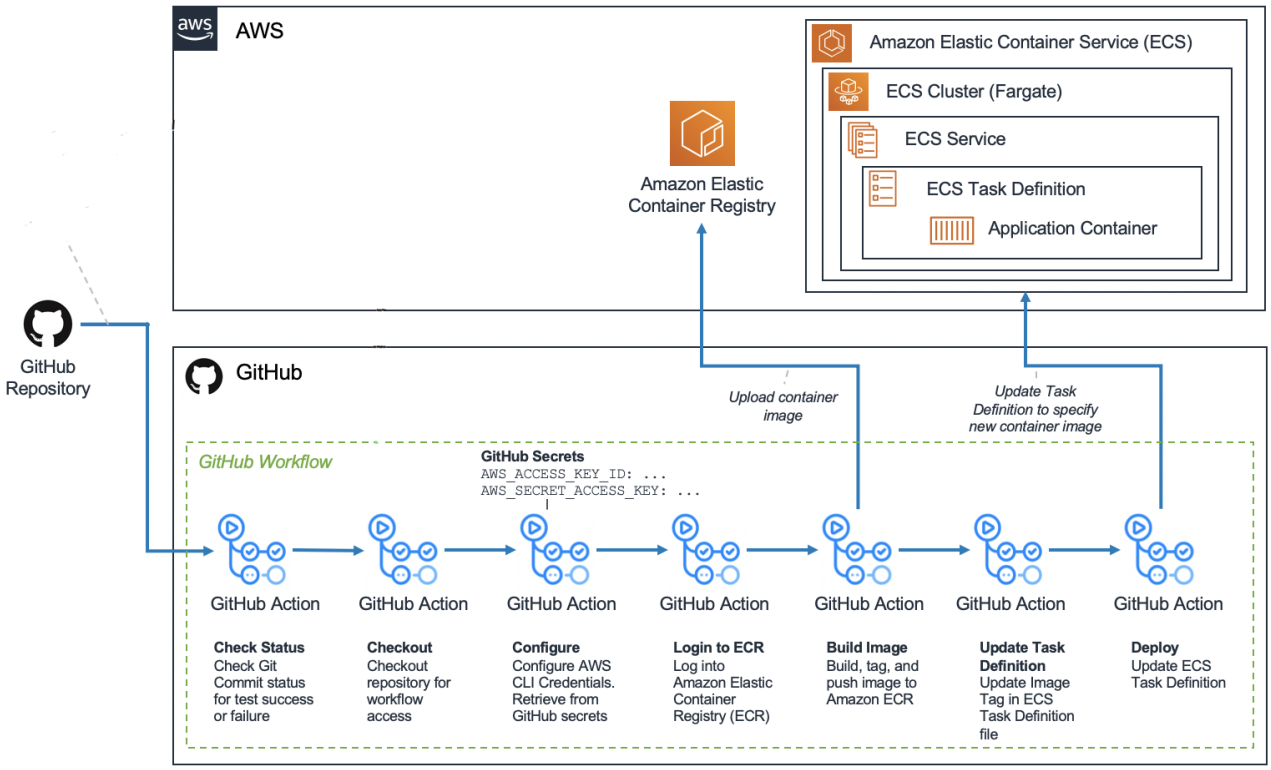
this time the worklow was successfull

Next

Now that we have our container ready and working and we understand how tests and github actions work, we can proceed to get things deployed on aws and setup a pipeline for it to be deployed automatically after running all the tests

the next steps are

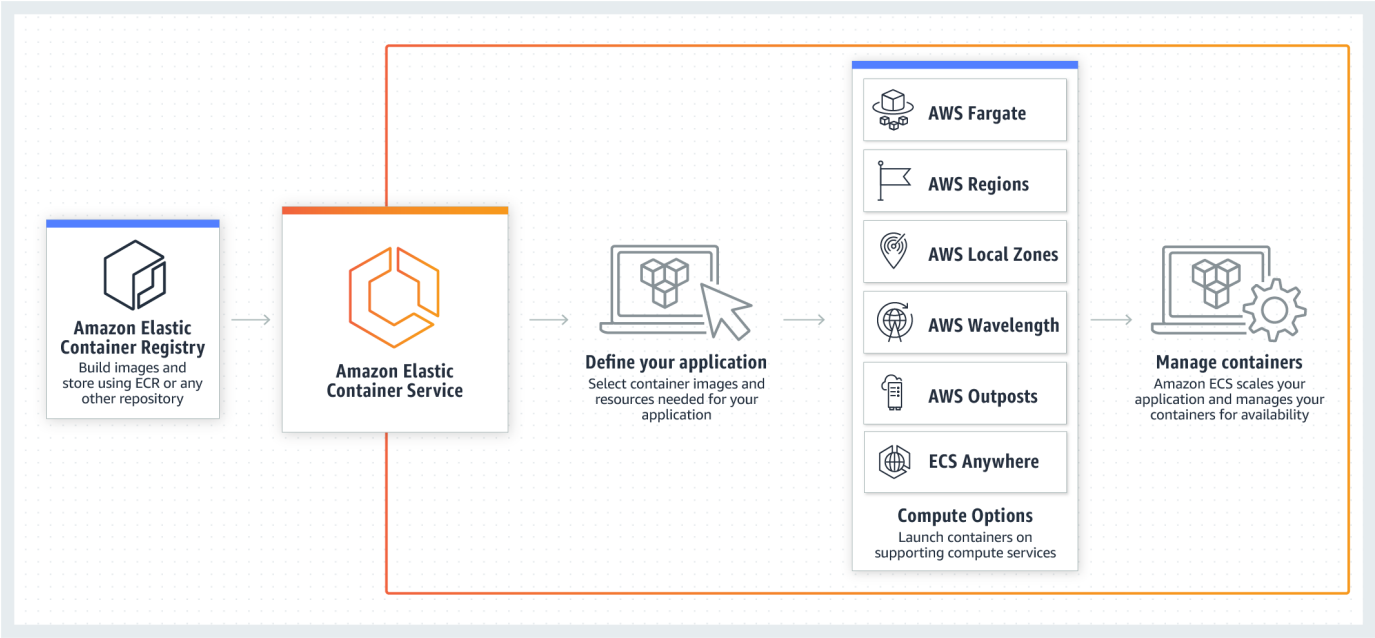
- to setup our aws account
- setup aws cli to push ar container to ecr (we will look into each service in a while)
- build and push the docker container to ecr
- setup our clusters, services and task for running the container
- check and run everything
- get back to our github workflow and add required actions to push the content from github to ecs for automatic deployment.



Deployment to AWS ECS

ECS - Elsatic Container Service

Amazon Elastic Container Service (ECS) is a cloud computing service in Amazon Web Services (AWS) that manages containers and lets developers run applications in the cloud without having to configure an environment for the code to run in. It enables developers with AWS accounts to deploy and manage scalable applications that run on groups of servers called clusters through API calls and task definitions.



ECR - Elastic container repository

Amazon Elastic Container Registry (Amazon ECR) is an AWS managed container image registry service that is secure, scalable, and reliable. Amazon ECR supports private repositories with resource-based permissions using AWS IAM.

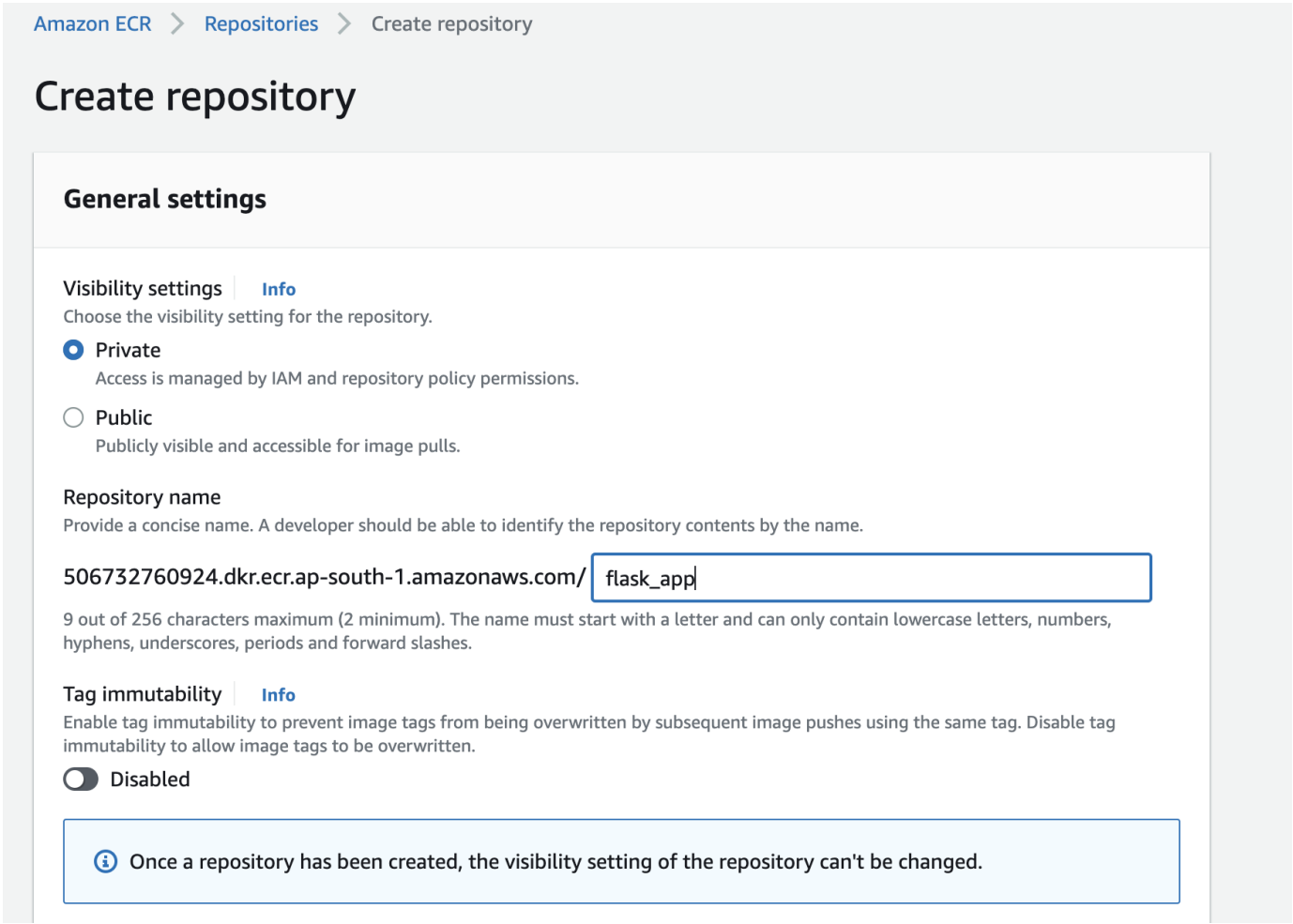
this is where we will store our docker image, and all other services will retrive the resources and docker images form here itslef.

To create an image repository

A repository is where you store your Docker or Open Container Initiative (OCI) images in Amazon ECR. Each time you push or pull an image from Amazon ECR, you specify the repository and the registry location which informs where to push the image to or where to pull it from.

Open the Amazon ECR console at https://console.aws.amazon.com/ecr/.

- Choose Get Started.
- For Visibility settings, choose Private.
- For Repository name, specify a name for the repository.
- For Tag immutability, choose the tag mutability setting for the repository. Repositories configured with immutable tags will prevent image tags from being overwritten. do not use this for now
- leave the rest as it is and click on create repository



after youn create the repository you will be back on the repository listing page

It will show the list of the repo. Click the flask_app repo, or the name you choose. it will open an images page that lists all the images in that repository, but it will be empty now, because we do not have any image till now.

> we will now create and push a docker image for the flask app that we have.

Build and check docker image locally

docker file:

```
# syntax=docker/dockerfile:1

FROM python:3.8-slim-buster

WORKDIR /flask-docker

RUN python3 -m pip install --upgrade pip
COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt
```

```
COPY . .

CMD ["python", "-m", "flask", "run", "--host=0.0.0.0"]
```

inside the folder with the required files and the docker file use the following command. the platform rg is important if you are using and arm pwered device like the m1 mac else you can skip that argument in the command

```
docker build --platform=linux/amd64 --tag=flask_app .
```

```
(flask_ecs) abdulahad_scaler@Abduls-Air Flask_ecs % docker build --platform=linu
x/amd64 --tag=flask_app .
[+] Building 290.2s (15/15) FINISHED
 => [internal] load build definition from Dockerfile                       0.0s
 => => transferring dockerfile: 167B                                       0.0s
 => [internal] load .dockerignore                                          0.0s
 => => transferring context: 2B                                            0.0s
 => resolve image config for docker.io/docker/dockerfile:1                 3.9s
 => CACHED docker-image://docker.io/docker/dockerfile:1@sha256:9ba7531bd8  0.0s
 => [internal] load .dockerignore                                          0.0s
 => [internal] load build definition from Dockerfile                       0.0s
 => [internal] load metadata for docker.io/library/python:3.8-slim-buster  3.2s
 => [1/6] FROM docker.io/library/python:3.8-slim-buster@sha256:db90d6607  38.3s
 => => resolve docker.io/library/python:3.8-slim-buster@sha256:db90d6607a  0.0s
 => => sha256:77c3488ce5476695aa0b018e6dcfe6c4a2bf2e5d1e4 1.37kB / 1.37kB  0.0s
 => => sha256:5cc94b67760dc542cb56567c7d7f5d9d19d9ec4bf6d 7.53kB / 7.53kB  0.0s
 => => sha256:32820e52a00eb22dc76d70d992be7082cd24b636 27.14MB / 27.14MB  36.6s
 => => sha256:6563993b0507bc602c27e376c26abd7766a8c8fa09 2.78MB / 2.78MB  11.6s
 => => sha256:eae6fe00ee3c3d4a62faa08362ff949fcf2fcb41 11.29MB / 11.29MB  23.0s
 => => sha256:db90d6607a1c2ccb9bbc8a6fdf286d183c44bf8e9c92193 988B / 988B  0.0s
 => => sha256:bbcf4b417983bad30035c8d28bae1dbca13f8e7636def1 234B / 234B  12.2s
 => => sha256:b2f1e8a99da34fd5fdcb38fcf88bd78ab7d248f3d2 3.18MB / 3.18MB  20.2s
 => => extracting sha256:32820e52a00eb22dc76d70d992be7082cd24b636cfb597ff  0.9s
 => => extracting sha256:6563993b0507bc602c27e376c26abd7766a8c8fa094da83b  0.1s
```

> additionally you can run and check the app locally

now we need to push this to ecr repo

AWS Permissions and access

- ECS full access
- ECR full access
- Cloud Watch
- cloud Formation

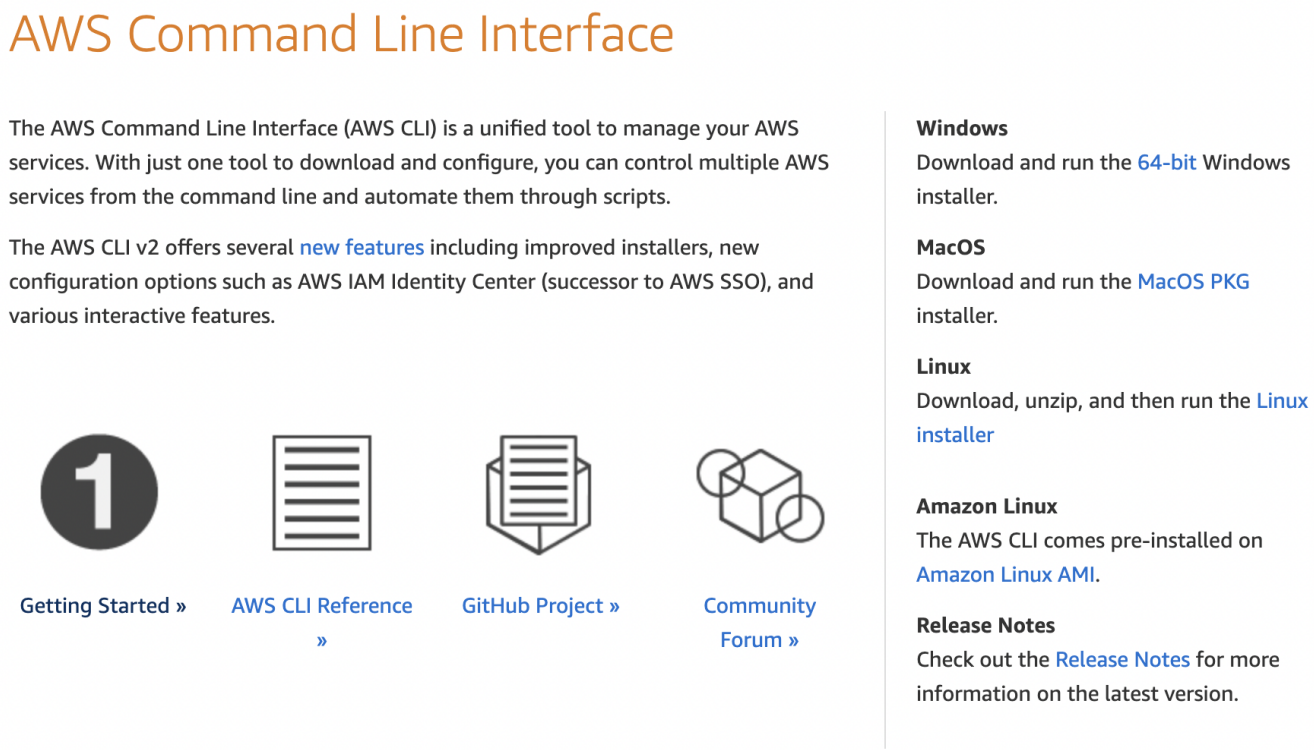
AWS CLI

Install the AWS CLI

You can use the AWS command line tools to issue commands at your system's command line to perform Amazon ECR and other AWS tasks. This can be faster and more convenient than using the console. The command line tools are also useful for building scripts that perform AWS tasks.

https://aws.amazon.com/cli/

according to you system use the above link to install the cli



Push docker to ECR

Build, tag, and push a Docker image

You use the Docker CLI to tag an existing local image and then push the tagged image to your Amazon ECR registry.

- Select the repository you created and choose View push commands to view the steps to push an image to your new repository.
- Run the login command that authenticates your Docker client to your registry by using the command from the console in a terminal window. This command provides an authorization token that is valid for 12 hours.

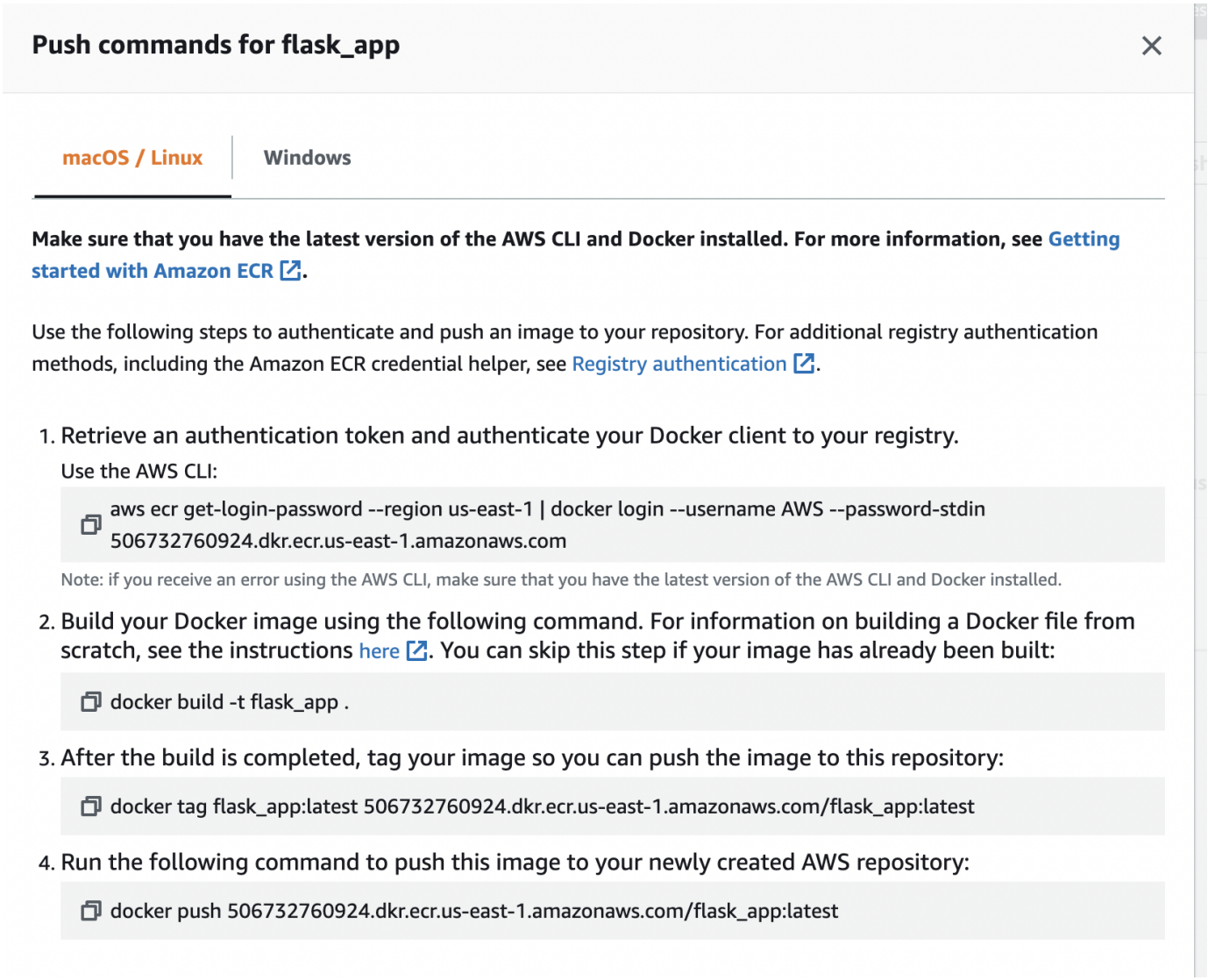
use the following command

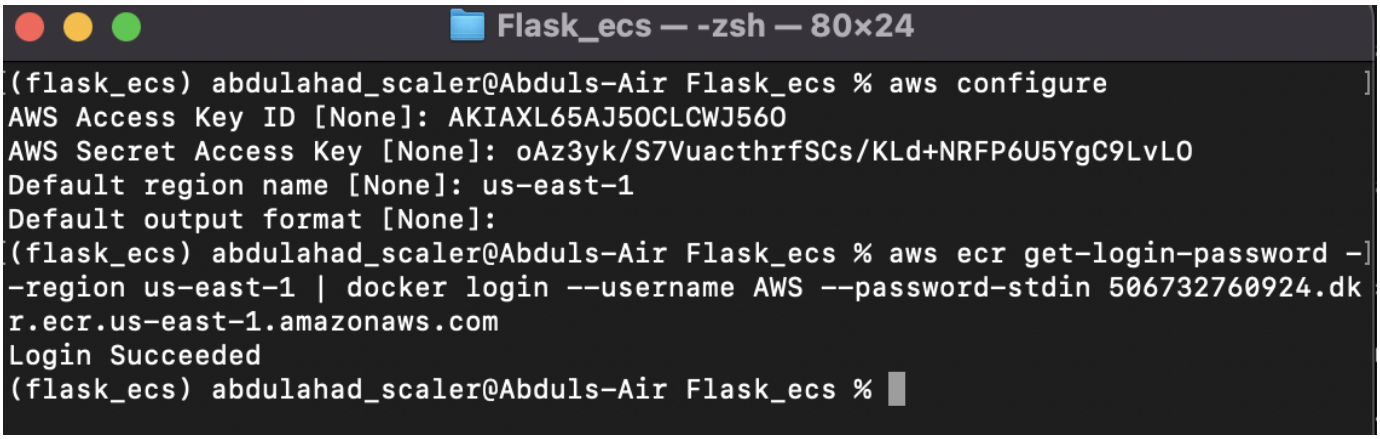
```
aws configure
AWS Access Key ID [None]: AKIAIOSFODNN7EXAMPLE
AWS Secret Access Key [None]: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

```
Default region name [None]: us-east-1
Default output format [None]:
```

after this you can use the push commands from the repository to move forward.

directly copy those commands and run them, you dont need to change anything





```
(flask_ecs) abdulahad_scaler@Abduls-Air Flask_ecs % docker push 506732760924.dkr
.ecr.us-east-1.amazonaws.com/flask_app:latest
The push refers to repository [506732760924.dkr.ecr.us-east-1.amazonaws.com/flas
k_app]
463880ed5ad1: Pushed
fd054f25fb1e: Pushed
32dc62f1e25e: Pushed
9edf2546b63c: Pushed
d457755e6d31: Pushed
7f2102a09744: Pushed
6ba5538cb764: Pushed
ffc2134468e0: Pushed
a332d6832c82: Pushed
3ebd45c2fd2c: Pushed
latest: digest: sha256:76e19987b1215a603dfe9564649055f0d572efef5a9ab57400ae184e1
d76739a size: 2421
(flask_ecs) abdulahad_scaler@Abduls-Air Flask_ecs %
```

Create a Fargate Cluster.

AWS Fargate

AWS Fargate is a serverless, pay-as-you-go compute engine that lets you focus on building applications without managing servers, it ia a serverless compute for containers, which eliminates the need to configure and manage control plane, nodes, and instances.

Let's return to the AWS management console for this step.

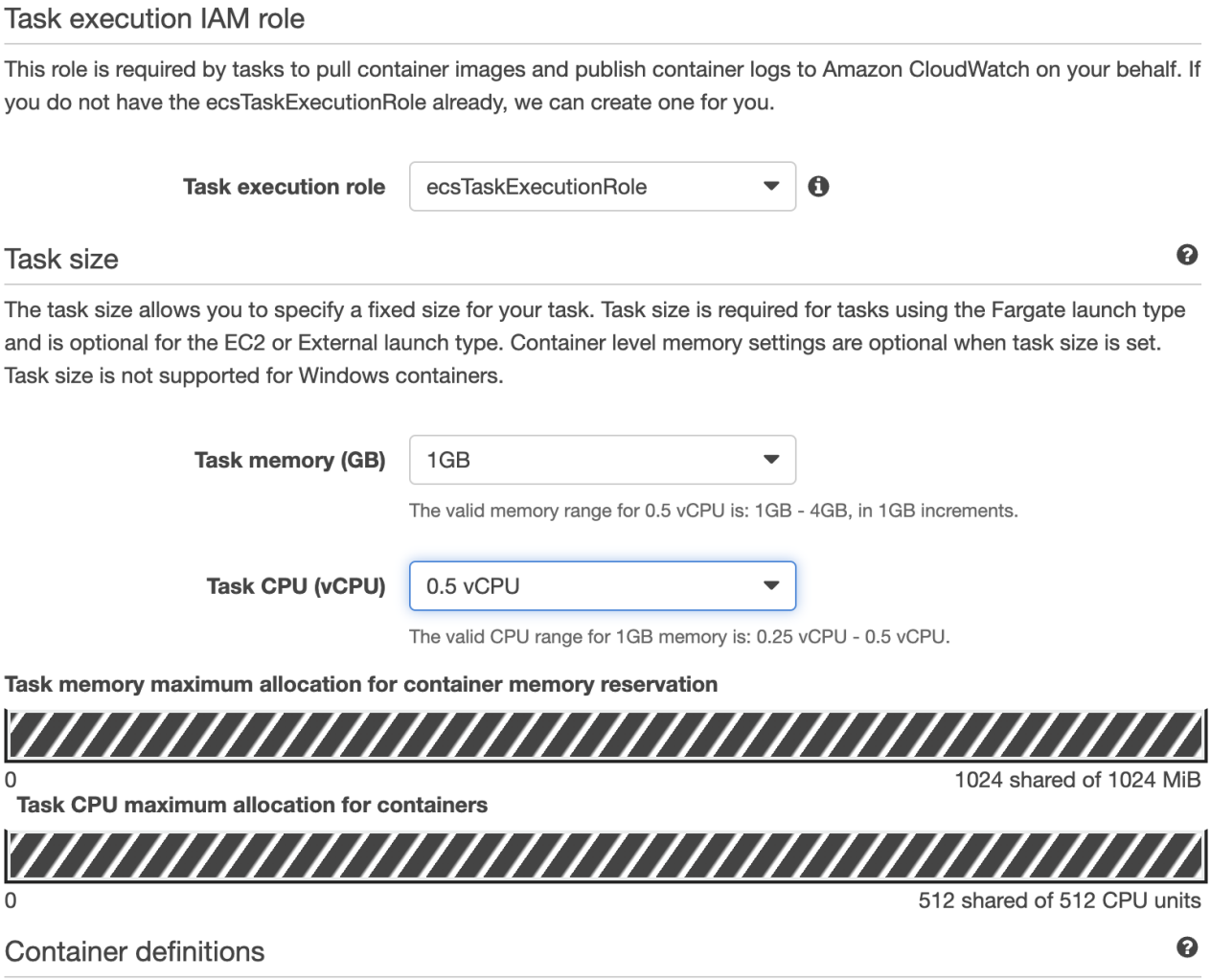
- Search for Elastic Container Service and select Elastic Container Service.
- From the left menu select Clusters
- Select Create cluster
- Under Select cluster template we are going to select networking only. We don't need ec2 instances in our cluster because Fargate will take care of spinning up compute resources when we start our task and spinning them down when we stop our task.
- we'll name the cluster default, and the rest we can leave as is.

Create an ECS Task

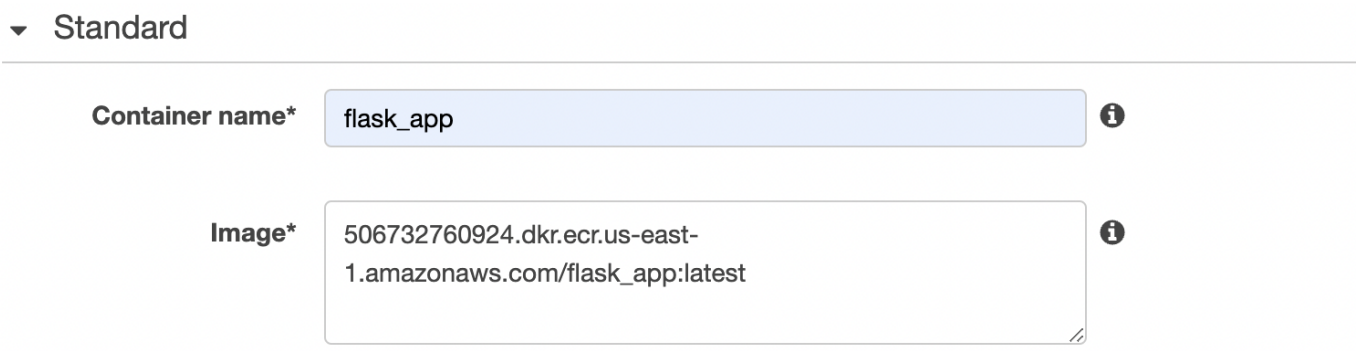
The ECS Task is the action that takes our image and deploys it to a container. To create an ECS Task lets go back to the ECS page and do the following:

- Select Task Definitions from the left menu.
- Then select Create new Task Definition
- Select Fargate
- Select Next Step
- Enter a name for the task. I am going to use flask_app.
- Leave Task Role and Network Mode set to their default values.
- Leave Task Execution Role set to its default.

- For Task memory and Task CPU select 0.5 v cpu and 1 gb memory. that will be sufficient for our usecase



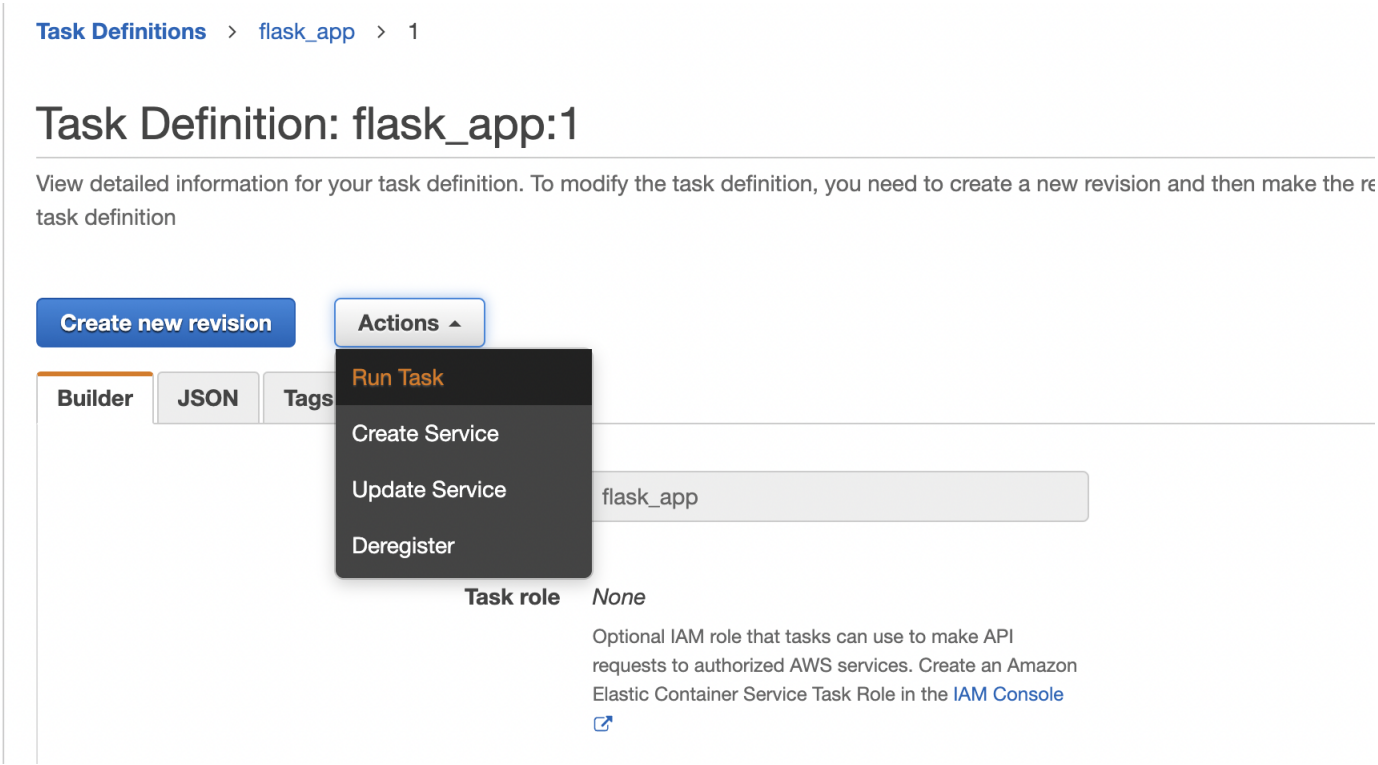
- Under Container definition select Add Container.
- Enter a Container name. we will use flask_app. In the Image box enter the ARN of our image. You will want to copy and paste this from the ECR dashboard .
- In port mappings you will notice that we can't actually map anything. Whatever port we enter here will be opened on the instance and will map to the same port on container. We will use 5000 because that is where our flask app listens.

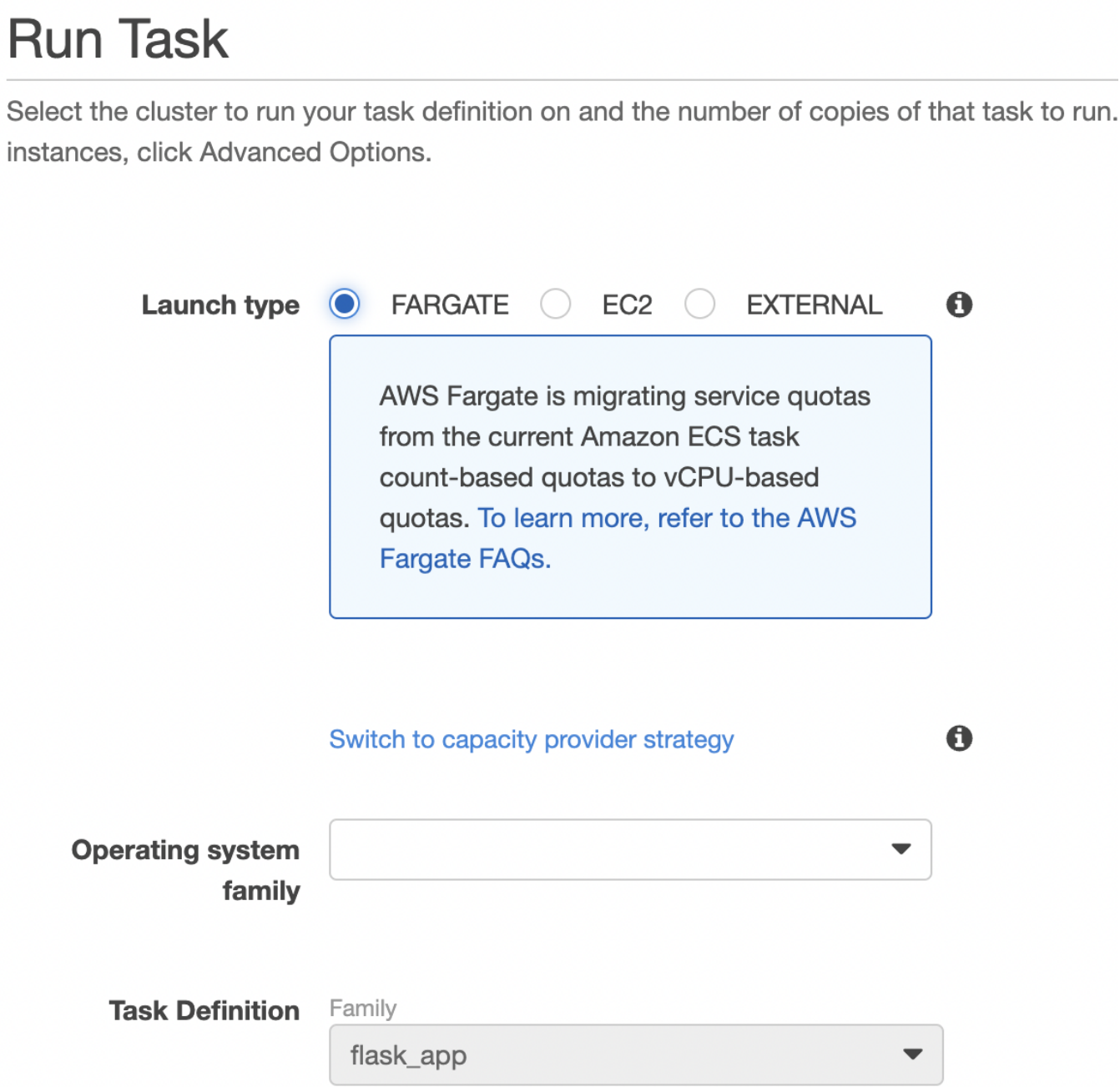


- Leave everything else set to its default value and click Add in the lower left corner of the dialog.
- Leave everything else in the Configure task and container definitions page as is and select Create in the lower left corner of the page.
- Go back to the ECS page, select Task Definitions and we should see our new task with a status of ACTIVE.

Run the ECS Task!

- Select the task in the Task definition list
- Click Actions and select Run Task



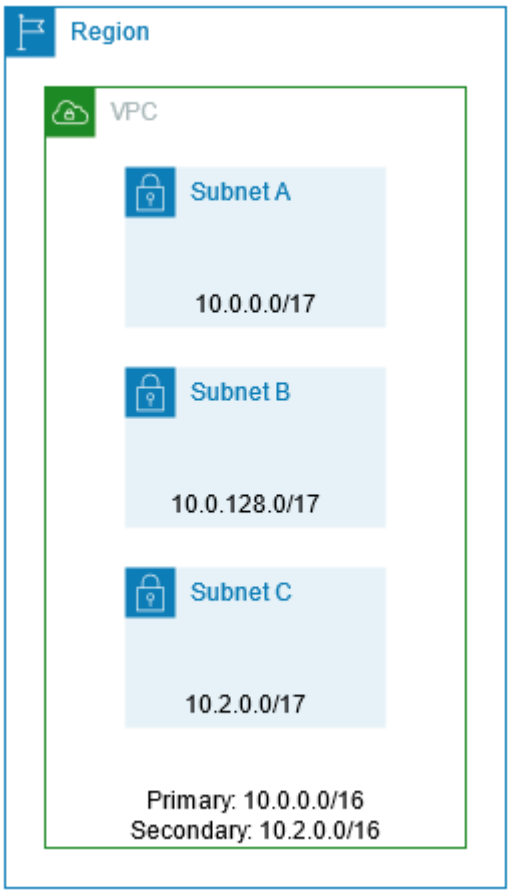


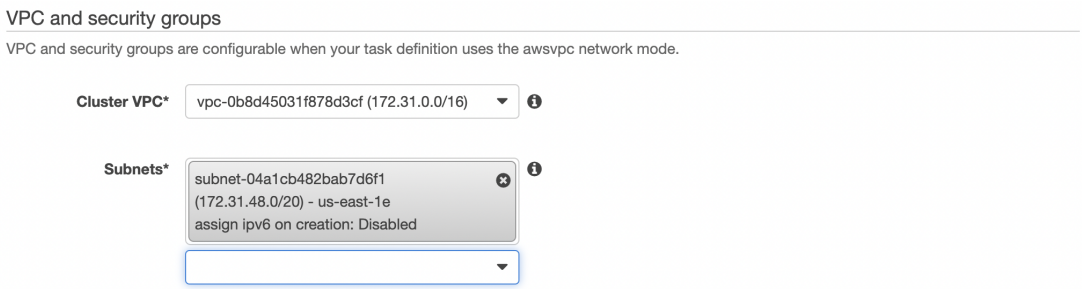
- For Launch type: select Fargate
- Make sure Cluseter: is set to the default we created earlier.
- Cluster VPC select a vpc from the list.
- Add at least one subnet.
- Auto-assign public IP should be set to ENBABLED

VPC and subnets

A virtual private cloud (VPC) is a virtual network dedicated to your AWS account. It is logically isolated from other virtual networks in the AWS Cloud. You can specify an IP address range for the VPC, add subnets, add gateways, and associate security groups.

A subnet is a range of IP addresses in your VPC. You launch AWS resources, such as Amazon EC2 instances, into your subnets. You can connect a subnet to the internet, other VPCs, and your own data centers, and route traffic to and from your subnets using route tables.

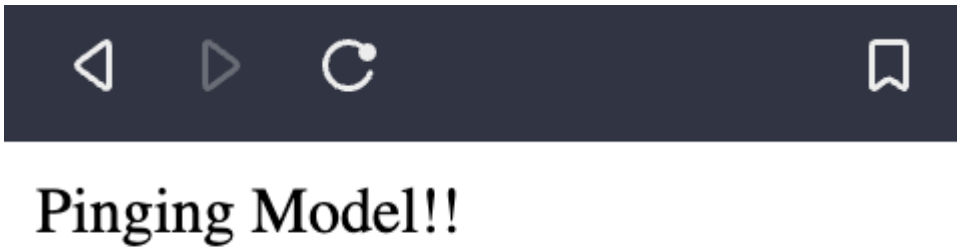




> Edit the security group. Because our app listens on port 5000, and we opened port 5000 on our container, we also need to open port 5000 in the security group. By default the security group created by Run Task only allows incoming connections on port 80. Click on Edit next to the security group name and add a Custom TCP rule that opens port 5000.

After you run the Task, you will be forwarded to the fargate-cluster page. When the Last Status for your cluster changes to RUNNING, your app is up and running. You may have to refresh the table a couple of times before the status is RUNNING. This can take a few minutes.

- click on the task
- Find the Public IP address in the Network section of the Task page.
- Enter the public IP address followed by :5000/ping in your browser to see your app in action.



> Now we have the app running we can also send our data to the app to see if everything is woring fine

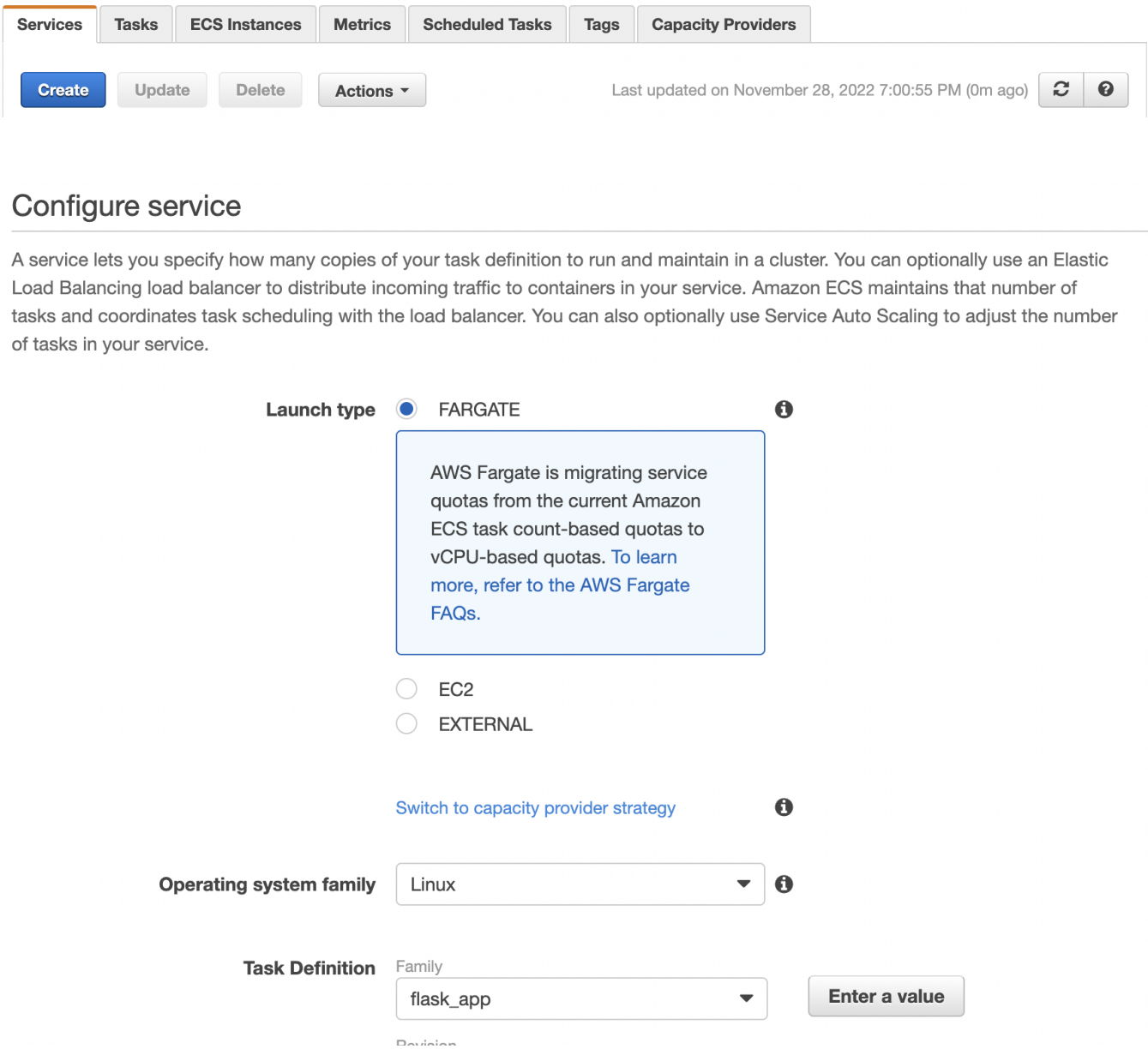
Setup the service as well

You can use an Amazon ECS service to run and maintain a specified number of instances of a task definition simultaneously in an Amazon ECS cluster. If one of your tasks fails or stops, the Amazon ECS service scheduler launches another instance of your task definition to replace it. This helps maintain your desired number of tasks in the service.

You can also optionally run your service behind a load balancer. The load balancer distributes traffic across the tasks that are associated with the service.

we will create a new service to handle our deployment and updates smoothly

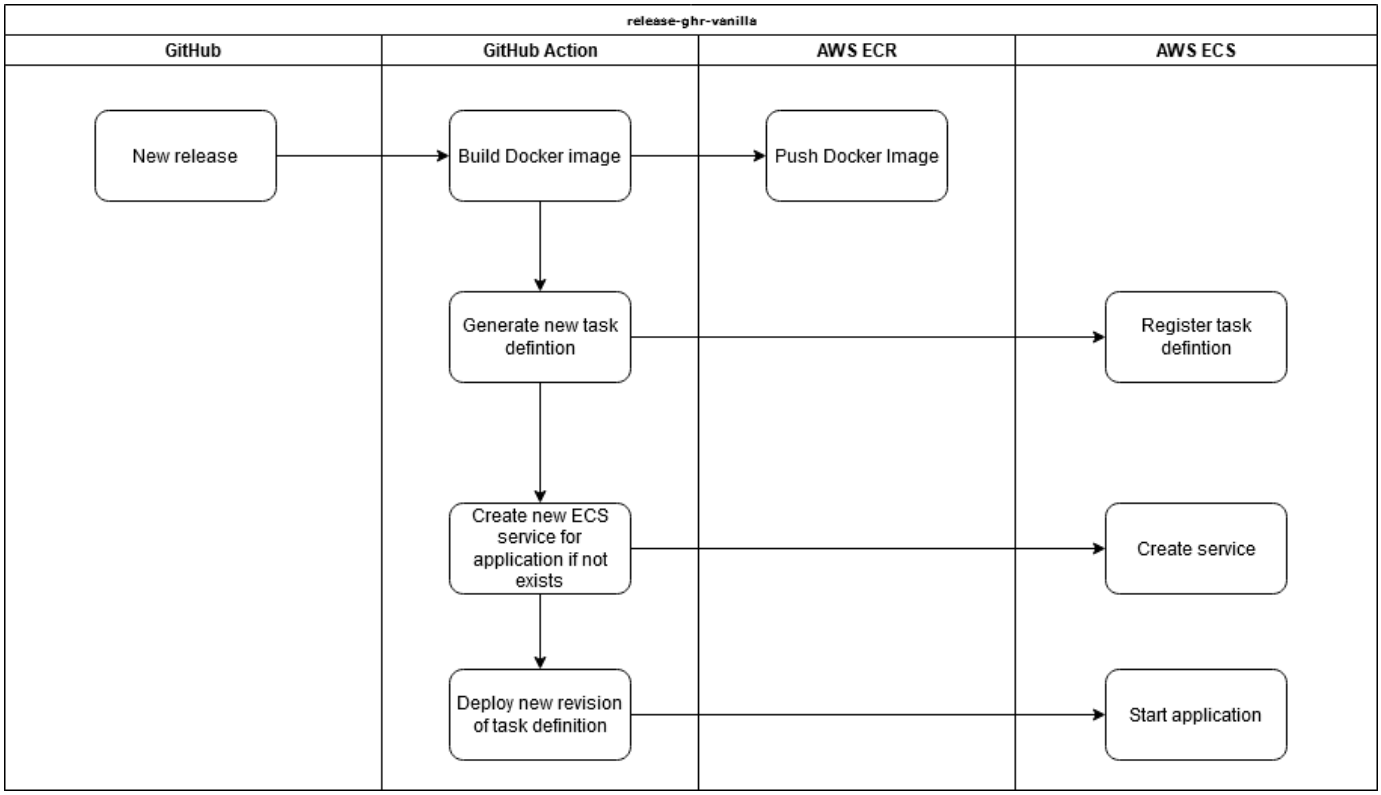
- select launch type as fargate
- select the task defination
- set a service name and you can leave all other things as default



Set up CD from github actions

Now that we have our app running on the ecs cluster we can now set up the cd pipeline through github actions so that everytime the repository on github gets updated we can directly push everything to ecr and deploy on ecs automatically

- we will agin use the github actions workflow to accomplish this



setup github for CD

> save the required credetials in the github account

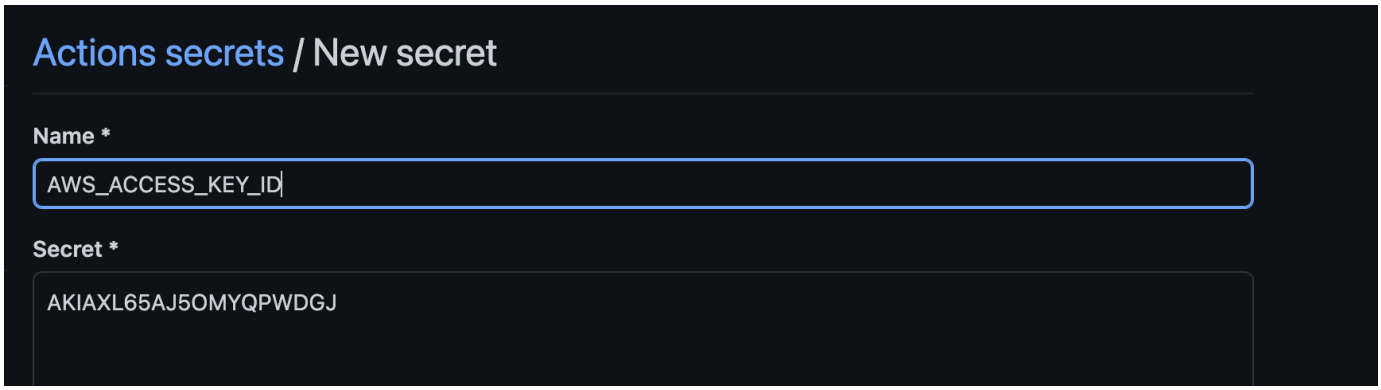
Secrets are encrypted environment variables that you create in an organization, repository, or repository environment. The secrets that you create are available to use in GitHub Actions workflows.

To create secrets for a personal account repository, you must be the repository owner. To create secrets for an organization repository, you must have admin access.

- On GitHub.com, navigate to the main page of the repository.
- Under your repository name, click Settings.
- In the "Security" section of the sidebar, select Secrets, then click Actions.
- Click New repository secret.
- Type a name for your secret in the Name input box.
- Enter the value for your secret.
- Click Add secret.

> we need to add two secrets

> - aws_access_key_id
> - aws_secret_access_key



We will add to the workflow file that we already created

> we can create a new workflow file, but we want to run this workflow job after the tests succeed so we will add this blow that and mark that this one need to other one to be successfull first

we'll name the first one job1 and in the second job add job2 needs job 1, this will make them run sequentially. by default they will run parallel that we do not want

- we are going to use some predefined actions already created for us by aws for our task.
- we just need to update the enviornment variables for it
- we also need the task defination that we created earlier so download the json file from the console
  - open the task defination click on json and copy the json from there and copy in a new file locally
- we will use different preconfigured actions for different talks like pushing to ecr and deploying to ecs
- we do not have to worry about any of them we just need to fill in the enviornment variables
- to know more about any workflow we can directly search about it. https://github.com/aws-actions

Environment Variables

- AWS_REGION — Operating region of AWS services.
- ECR_REPOSITORY — Name of the ECR repository that you have created.
- ECS_SERVICE — Service name of the ECS Cluster.
- ECS_CLUSTER — Name of the ECS Cluster.
- ECS_TASK_DEFINITION — Path of the ECS task definition in JSON format which is stored in GitHub repository.

- CONTAINER_NAME — Docker container name under the ECS task definition.

```
name: Deploy to Amazon ECS

on:
  push:
    branches:
      - main

env:
  AWS_REGION: us-east-1                   # set this to your preferred AWS region, e.g. us
  ECR_REPOSITORY: flask_app           # set this to your Amazon ECR repository name
  ECS_SERVICE: flask_app_service                 # set this to your Amazon ECS service nam
  ECS_CLUSTER:  default                  # set this to your Amazon ECS cluster name
  ECS_TASK_DEFINITION: task-definition.json # set this to the path to your Amazon ECS task
                                            # file, e.g. .aws/task-definition.json
  CONTAINER_NAME: flask_app           # set this to the name of the container in the
                                            # containerDefinitions section of your task


jobs:
  job1:
    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



  job2:
    needs: job1
    runs-on: ubuntu-latest
    environment: production
```

```
    steps:
      - name: Checkout
        uses: actions/checkout@v3

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@13d241b293754004c80624b5567555c4a39ffb
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ env.AWS_REGION }}

      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@aaf69d68aa3fb14c1d5a6be9ac61fe15b48453a2

      - name: Build, tag, and push image to Amazon ECR
        id: build-image
        env:
          ECR_REGISTRY: ${{ steps.login-ecr.outputs.registry }}
          IMAGE_TAG: ${{ github.sha }}
        run: |
          # Build a docker container and
          # push it to ECR so that it can
          # be deployed to ECS.
          docker build -t $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG .
          docker push $ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG
          echo "image=$ECR_REGISTRY/$ECR_REPOSITORY:$IMAGE_TAG" >> $GITHUB_OUTPUT

      - name: Fill in the new image ID in the Amazon ECS task definition
        id: task-def
        uses: aws-actions/amazon-ecs-render-task-definition@97587c9d45a4930bf0e3da8dd2feb2
        with:
          task-definition: ${{ env.ECS_TASK_DEFINITION }}
          container-name: ${{ env.CONTAINER_NAME }}
          image: ${{ steps.build-image.outputs.image }}

      - name: Deploy Amazon ECS task definition
        uses: aws-actions/amazon-ecs-deploy-task-definition@de0132cf8cdedb79975c6d42b77eb7
        with:
          task-definition: ${{ steps.task-def.outputs.task-definition }}
          service: ${{ env.ECS_SERVICE }}
          cluster: ${{ env.ECS_CLUSTER }}
          wait-for-service-stability: true
```

- all the aws actions are in job 2
- it will run after job 1 runs successfully because we mentioned job 2 needs job1