The Process of Containerising and Deploying a Flask Application

The general overview of deploying a flask app using ECS was as follows:

1. Write a flask app. The one I wrote can be found in the our Github repository.
2. Containerise the flask app with Docker. This involves writing a Docker file (also on Github) and building the Docker image of the application.
3. Deploying the Docker image on ECS. This involves pushing the the image onto an AWS repository (ECR), creating an ECS cluster, creating a *class definition*, and running the class definition on the ECS cluster.

Stage 1 - Writing a Flask App

Main things to consider are making sure Python and a pip installer are installed on your machine. From there you can create a new project directory and a virtual environment with

```
mkdir projectFolder && cd projectFolder
python3 -m venv ./venv
```

We can then activate the virtual environment with

```
source ./venv/bin/activate
```

(To later deactivate the virtual environment we can do `deactivate`)

So now all python installations will be specific to this virtual environment.

Run `pip3 install Flask`

For **Windows** commands, they can be found here Installation — Flask Documentation (3.1.x)

From there, we can write the Flask application. A simple app will have only one route. Create a file **app.py** and the following code.

app.py > ...

```python
from flask import Flask, render_template
app = Flask(__name__)

@app.route('/')
def index():
    # only need the file name because render_template knows to look in the directory called "templates"
    return render_template('index.html')

if __name__ == "__main__":
    app.run(host ='0.0.0.0', port=5001, debug = True)

```

Code for app.py

An important thing to note is that the **host='0.0.0.0'**.

By following a Flask tutorial, I also saw how to render html files with a Flask app. This is outside the scope of this document, but in essence, you create a directory called **templates** and an html file (**index.html**) in templates. In the **app.py** file, we import render_template from flask and use this function to render our HTML file. The exact structure of these files can be seen in the Github repository. If not interested in rendering an html file, replace `render_template(index.html)` with `"Hello World"`

Stage 2 - Creating a Docker Image of the Flask App

I would recommend downloading the **Docker** application from Docker Desktop: The #1 Containerization Tool for Developers | Docker to make it easy to visualise what is going on when you containerise the Flask app. I believe you are able to do everything from the terminal and the application is quite heavy (~1.8GB) so this step is optional.

Create a file **requirements.txt** in your working directory. This is where your python packages go. We only have flask, so that is all we have to write.

```
requirements.txt
1    flask
```

Create a **Dockerfile**. Be sure to name it exactly “Dockerfile”. This file is the instructions to run when building your docker image. The particular Dockerfile I have is as follows

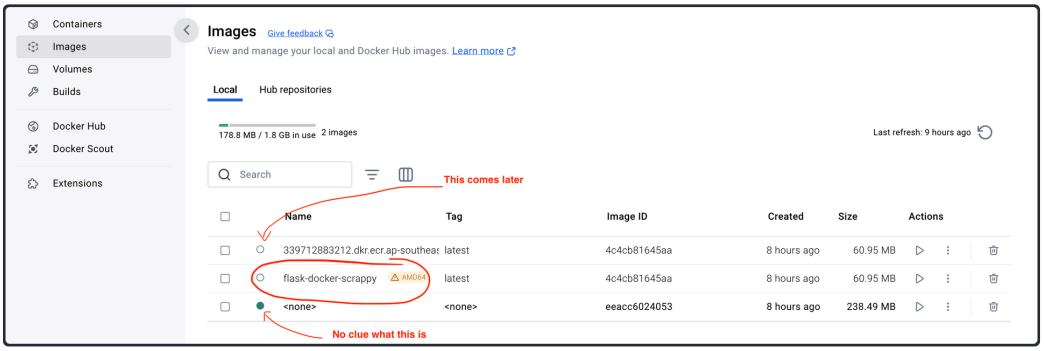
```
1   FROM python:3.8-slim-buster
2
3   WORKDIR /python-docker
4
5   COPY requirements.txt requirements.txt
6   RUN pip3 install -r requirements.txt
7   EXPOSE 5001
8   COPY . /python-docker
9   ENTRYPOINT [ "python3" ]
10  CMD [ "app.py" ]
```

We now run the **build** command:

```
docker build -t flask-docker-scrappy --platform linux/amd64 .
```

- The **-t** option is to give the docker image a *tag*
- The **--platform linux/amd64** part is to specify what kind of operating system we want the image to be working on. This part is *not* necessary for building the image, but was required when getting the image to run on AWS ECS. To save time, add it here.
- **.** is to specify the working directory as the

If you downloaded the Docker Desktop application, you should now see the following:

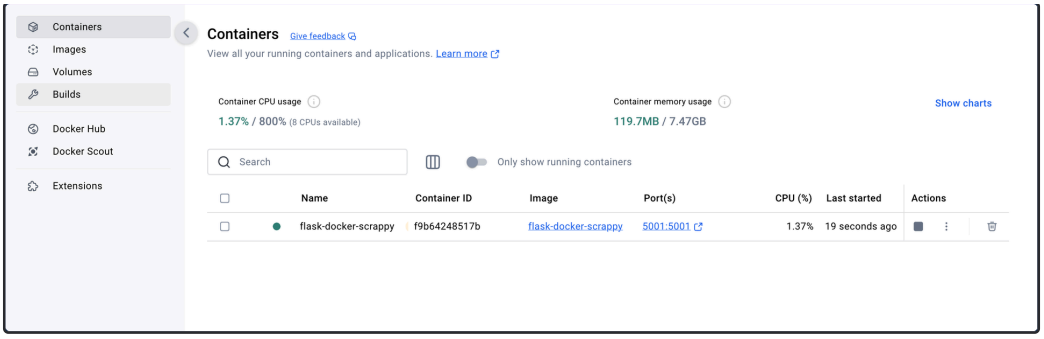


We can run the docker image (not necessary for deployment, but nice to see how it works) using the following **run** command

```
docker run --name flask-docker-scrappy -p 5001:5001 flask-docker-scrappy
```

- The **--name flask-docker-scrappy** part tells us what to name the *container*; we could name it whatever we like
- The **-p 5001:5001** part is for port forwarding; it is what allows us to access port 5001 on the Docker machine.
- We specify the image we want to run - in this case **flask-docker-scrappy**

You should now have something that looks like this in the Docker Desktop application



Clicking on the name of the container (flask-docker-scrappy) should take you to a terminal view

and clicking on the 127.0.0.1 address should give you access to the Flask app

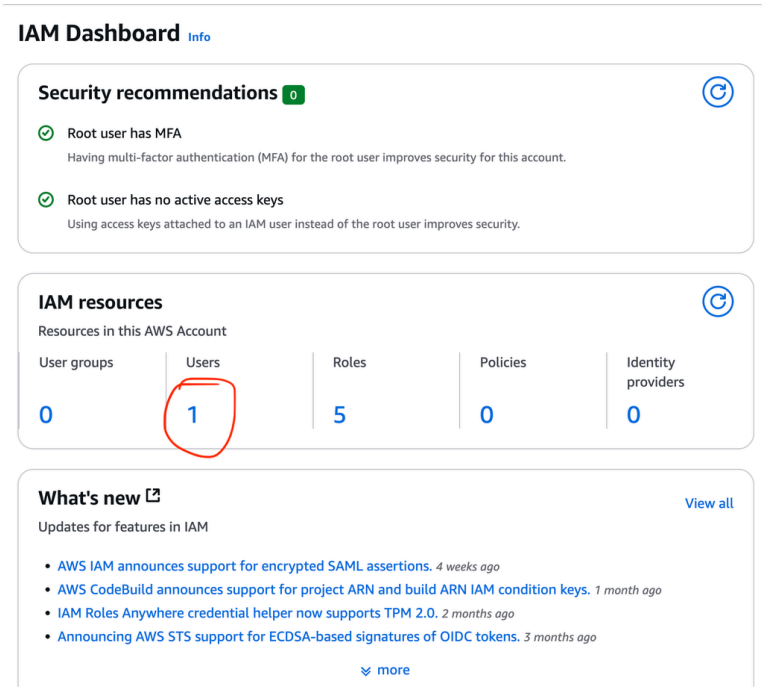
So this indicates that we have successfully built a Docker image.

Stage 3 Deploynig the Docker Image on ECS

Unfortunately, this part was annoying convoluted. There are a few AWS configuration/set up steps that have to be done. If not done already, create an AWS account. From there, we need to work with several services.

IAM

We need to create an IAM user to facilitate uploading our image to ECR. Go to the IAM service, click on the number under users and create a new user.



Click on this number in the IAM service

Name the user whatever you like. The important thing is to provide relevant permissions to the user. The permissions my user used were AdminstratorAccess, AmazonEC2ContainerRegistryFullAccess and AmazonECS_FullAccess. However I believe **AmazonEC2ContainerRegistryFullAccess** is the only one that is required, and the other two came from trying to fix a different problem and are actually not necessary fort his task.

The permissions I gave to my IAM user

Now that the IAM user has been created, we must download the AWS CLI from Installing or updating to the latest version of the AWS CLI - AWS Command Line Interface

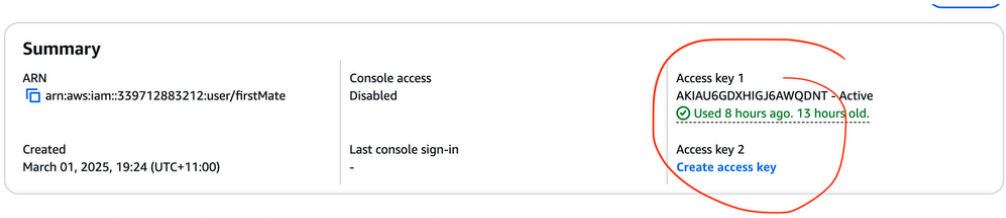
Once the AWS CLI has been installed, check it has been successful by running the following in your terminal

```
aws help
```

and you should get output like

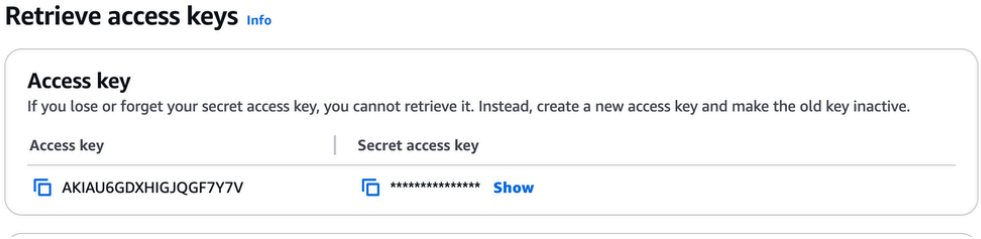
```
AWS()
NAME
       aws -
DESCRIPTION
       The AWS Command Line Interface is a unified tool to manage your AWS
       services.
```

From there, we need to connect our CLI to our IAM user. We need to create an **access key** for our new IAM user. Go to the IAM service again, click on the number under users, select your newly created IAM user and click **Create access key**.



"Create access key" should be under Access key 1; since I have created an access key, for me it is under Access key 2

Go through the process. By the end you should have this screen:



Both the access key and the secret access key will be required. Note that if you exit this page, you will **not** be able to retrieve the secret access key and will have to create a new access key.

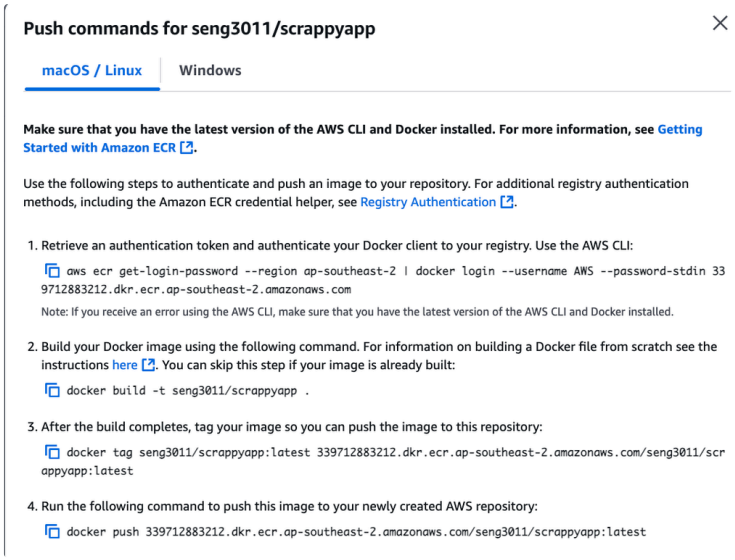
In your terminal, run

```
aws configure
```

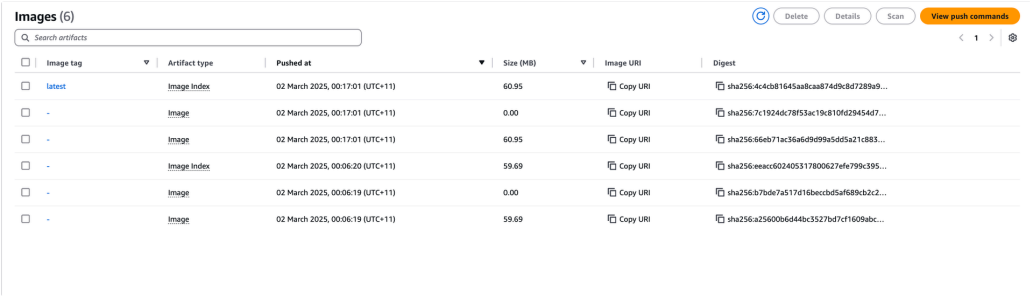
It will ask for an access key. Provide the one you created. It will then ask for a secret access key. Provide the one you created. Your AWS CLI is now being "used" by the IAM role.

ECR

This is where we will place our Docker image. Go to the ECR service and create a new **private repository**. Keep everything as default, provide a repository name. When you go back to ECR, you should be able to see your new repository under the **private repositories tab**. If you click on the repository, you should see a button called **push commands**.



Follow these instructions, and you should be able to push your Docker image to the repository.



Under the images tab - since I pushed twice I have 6 items; after pushing once you should get 3 items

Click on **Copy URI** for the **latest image index**. We will need this later.

ECS

Everything written in the ECS section is done in this tutorial: How to Deploy Flask App on AWS ECS using AWS Fargate?

Go to ECS, go to **Clusters** and create a new cluster. Provide a name for the cluster, keep Fargate selected for your infrastructure but keep everything else default.

Now go to **task definitions**. Create a new task definition. There are a few important configurations to make here.

Platform

Ensure **Linux/X86_64** is selected because this is the platform we specified the Docker image to be when we built it.

Task Execution Role

Ensure **Create new role** is selected so that a role with the necessary permissions to run this task is created

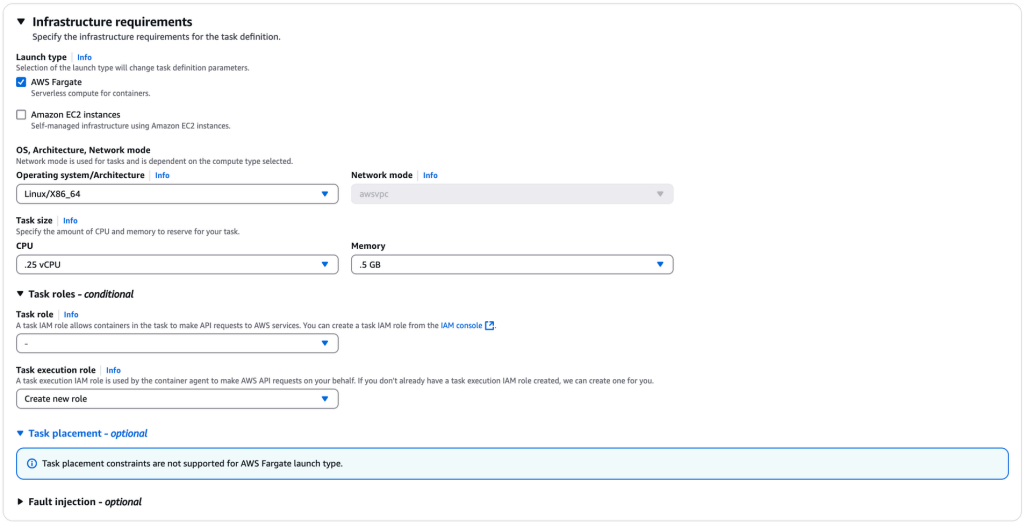
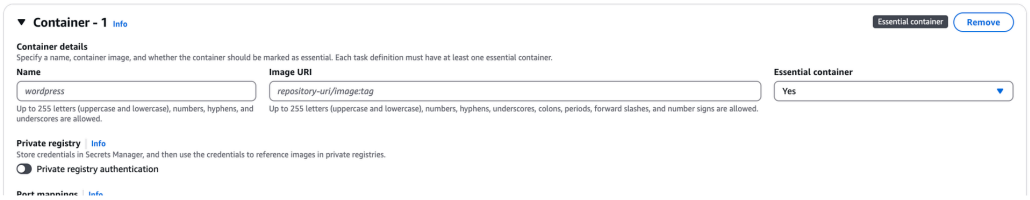


Image URI

Paste the image URI you previously copied (can be found in ECR) here. Give whatever name you like.

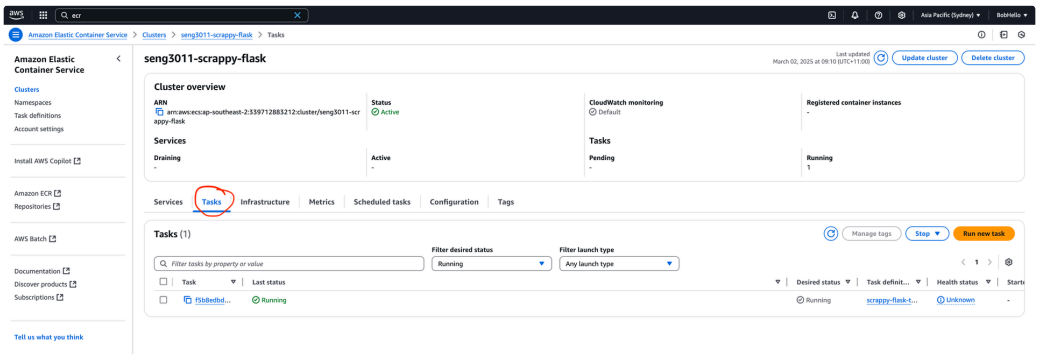


Port

Give the same port that you were running on the Flask app and the Docker image.

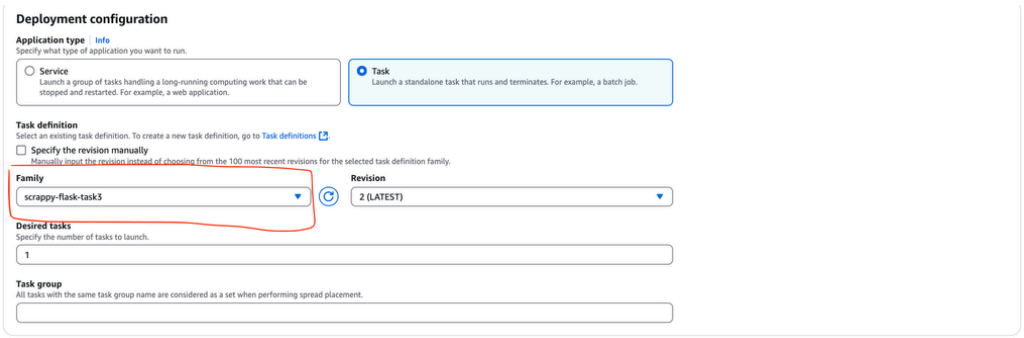
Everything else can be left as default. Create the task definition.

This task definition tells us that we *should* run the Docker image located in our ECR repository. However, the image is not actually running yet. We go back to our cluster, click on the **Tasks** tab and click **Run new task**.

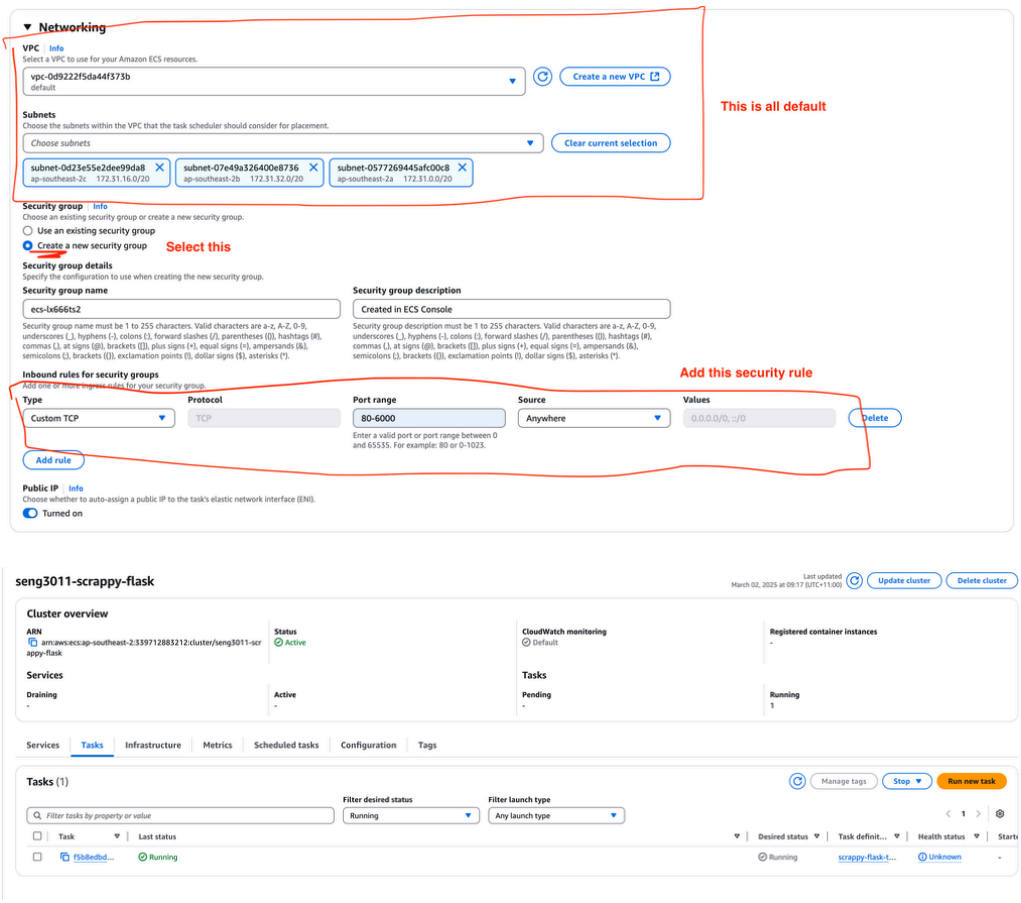


The tutorial I followed selected **Launch type** instead of **Capacity provider strategy** but I am not sure that makes a difference.

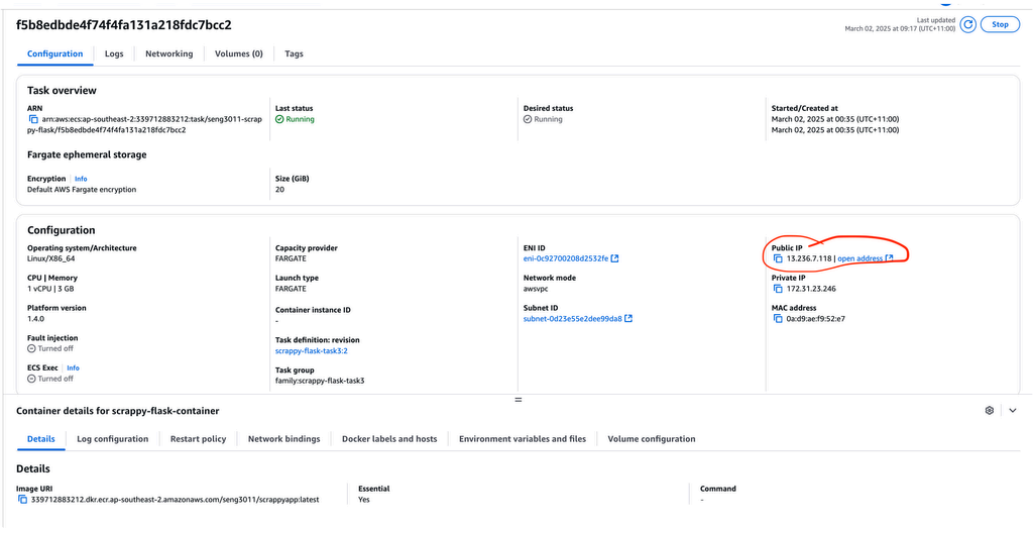
The two important things to do are providing the task definition under the **Family** heading in **Deployment configuration**



and under the **Networking** tab, making the task and the relevant port available from anywhere. Also make sure **Public IP** is turned on.



If you click on the cluster, and click on the new task (**f5b8edbd...** for me) you should be able to see a public IP address.

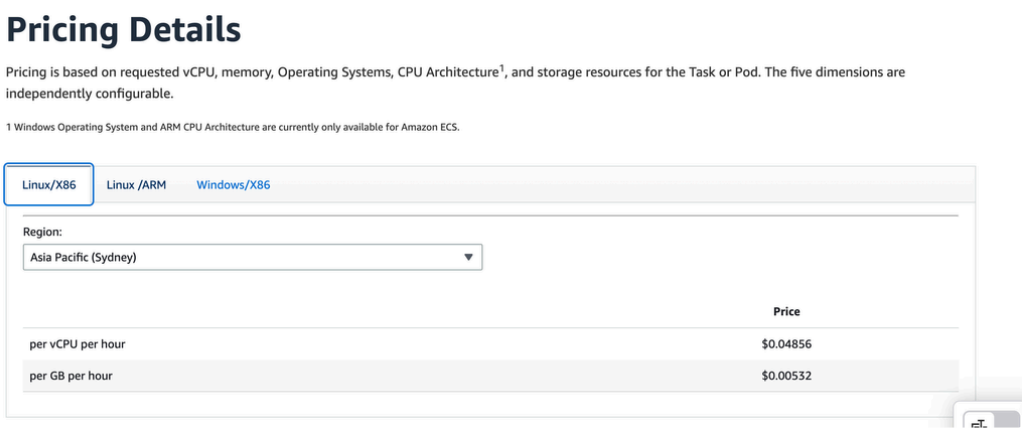


Public IP address of the task running on the cluster

If you take this IP address, add a **:** and the port number (for me 5001), you should have the address to access Flask app publicly. So for me, this address would look like http://13.236.7.118:5001/.

Cost Considerations

Depending on what kind of CPU and Memory you selected for creating your cluster, your costs will scale accordingly.



Fargate costs in Sydney

NOTE that using Linux/Arm is cheaper than Linux/X86.

ECR seems essentially free for a year because we would get 500 MB / month for the first year