Amazon Elastic Container Service (Amazon ECS)

SERVERLESS CONTAINERIZED WEATHER APP ON AWS ECS

Harvey Aland

March 28th, 2025

Introduction

In today's fast-paced technology landscape, the ability to build, deploy, and manage applications efficiently is crucial. This project demonstrates the end-to-end process of developing a containerized weather application using **Docker**, deploying it to **Amazon Elastic Container Registry (ECR)**, and running it on **Amazon Elastic Container Service (ECS)**. The infrastructure is provisioned using Terraform, showcasing the power of Infrastructure as Code (IaC).

This document provides a detailed walkthrough of the project, including the code, architecture, and deployment process. It also highlights the relevance of this project to both my personal growth and the broader industry.

Overview of the Project

The project involves building a simple weather application using Python and Flask, containerizing it with Docker, and deploying it to AWS using ECS and ECR. The infrastructure is provisioned using Terraform, which automates the creation of resources such as VPC, subnets, ECS clusters, and an Application Load Balancer (ALB).

Key Components

1. **Application:**
    - A Python-based weather app that retrieves and displays weather data.
    - Built using Flask for the backend and containerized using Docker.

2. **Infrastructure:**
    - VPC: A Virtual Private Cloud with public and private subnets for secure networking.
    - ECS: Elastic Container Service for running the containerized app.
    - ECR: Elastic Container Registry for storing Docker images.
    - ALB: Application Load Balancer for routing traffic to the ECS tasks.

3. **Tools and Technologies:**
    - Docker: For containerizing the application.
    - Terraform: For provisioning and managing AWS infrastructure.
    - AWS CLI: For interacting with AWS services.
    - GitHub: For version control and collaboration
    - Python: For basic flask app
    - HTML: Written front end content for app

Why This Project is Relevant

1. **Skill Development:**
    - This project allowed me to gain hands-on experience with Docker, AWS ECR, ECS, and Terraform.
    - I learned how to containerize an application and manage cloud infrastructure.

2. **Portfolio Building:**
    - This project serves as a tangible demonstration of my ability to design, build, and deploy a modern cloud-native application.

3. **Containerization:**
    - Containerization is a cornerstone of modern software development, enabling applications to run consistently across different environments. This project showcases my ability to leverage Docker for containerization.
    -
4. **Infrastructure as Code (IaC):**
    - Terraform is widely used for automating infrastructure provisioning. This project highlights my ability to use Terraform to manage cloud resources efficiently.

Flask application build

While researching what application I could create as a beginner to showcase containerizing an application in docker I found myself looking at different small apps like a simple message response or a shopping list. However, neither of which I felt had any real-world value.

I wanted to build something that I thought for the most part was cool and offered some form of practicality. So, I settled on a weather app since I rely on them daily, especially living in the unpredictable Northern English climate.

To get weather data, I used OpenWeatherMap.com, which provides a free API for region-specific forecasts. I went with London for my region of choice.

Firstly, I had to confirm that the API is working and outputting the data before I could build the app.

Checking API URL

Building the app using Python Flask

what is flask? - Flask is a lightweight and flexible web framework for python, it allowed me to build my web application quickly without needing a lot of boilerplate code. This was a viable option for me as I have experience using python and it was relatively straight forward for what I was building.

How Does Flask Work? - Flask runs a webserver that listens for requests from users, you can define routes (URLs) in python to determine what should happen when someone visits a specific page. For my project I chose flask to render a HTML page

Writing the Python Code

The Python script (app.py) does the following:

1. Initializes a Flask application.
2. Fetches real-time weather data from OpenWeatherMap for a specific city.
3. Defines routes for the application:
    - / → Displays the weather on the webpage.
    - /health → Used for health checks by AWS ALB.
4. Runs the app on port 5000 so it can be accessed inside the container.

The Python Flask code is annotated with notes which tells you what each line does. This script took a while to get right as it was my first time creating a flask app, I had to reference YouTube tutorials and a little help from AI too to get it right.

weather_app.py > health

```python
import requests  # This is like telling Python, "I want to use a tool to ask for information from the internet."
from flask import Flask, render_template  # Telling Python that we are going to make a web app with Flask and sh

app = Flask(__name__)  # This is creating a "Flask" app. Imagine it as a box where our web app lives.

API_KEY = 'd552c10073f57e64af138603e24430e7'  # This is like your secret password to access the weather data.
CITY = 'London'  # We want the weather for London.
BASE_URL = f'http://api.openweathermap.org/data/2.5/weather?q={CITY}&appid={API_KEY}&units=metric'  # This is th

#Health check
@app.route('/health')
def health():
    return "OK", 200  # ALB will receive a 200 status, meaning the app is healthy.


@app.route('/') # This means "When someone goes to our website's home page, do the following."
def index():  # This is the function that runs when someone visits the page.
    response = requests.get(BASE_URL)  # We ask the weather website for the data.
    data = response.json()  # We take the data the website gives us and change it into a format we can use.

    # Check if everything went fine with the data
    if data["cod"] == 200:  # If the website says it's all good (code 200 means everything worked)
        temperature = data["main"]["temp"]  # We get the temperature from the data.
        description = data["weather"][0]["description"]  # We get the weather description (like "sunny" or "clou
        city_name   = data["name"] # Extract the city name from the API response
    else:
        temperature = None  # If something went wrong, we say there's no temperature.
        description = "Error fetching data"  # We tell the user there was an error.

    return render_template('index.html', temperature=temperature, description=description, city=city_name)  # Sh

app.run(host="0.0.0.0", port=5000, debug=True)
```

After the flask code was done, I had to create the requirements.txt file since the flask app depends on certain python libraries.

requirements.txt

```
flask # Runs web application
requests # Allows your app to fetch weather data from the OpenWeatherMap API
gunicorn # Provides a more robust server for running Flask in Docker (instead of the built-in development server).
```

HTML template for the app frontend

Next step was to create a frontend page that users of the app would see when accessing the app.

Seeing as I've used HTML before this part was relatively easy, upon researching how HTML and Python flask can be used together I found a template that would be best suited for my project and did some tweaks. Since I only needed a simple layout, this template worked well with minimal modifications.

The index.html file is a Jinja template used by Flask to dynamically display weather data fetched from OpenWeatherMap. When a user visits the app, Flask retrieves weather information and passes it to the template using the render_template() function. The placeholders inside {{ }} are replaced with real values, such as the city name, temperature, and weather condition. This ensures that each time the page loads, it reflects the latest weather data. The template also includes simple CSS styling to format the output neatly, making the app visually appealing and easy to read.

Flask App Code that links with Jinga HTML template

```python
# Check if everything went fine with the data
if data["cod"] == 200:  # If the website says it's all good (code 200 means everything worked)
    temperature = data["main"]["temp"]  # We get the temperature from the data.
    description = data["weather"][0]["description"]  # We get the weather description (like "sunny" or "cloudy").
    city_name   = data["name"] # Extract the city name from the API response
else:
    temperature = None  # If something went wrong, we say there's no temperature.
    description = "Error fetching data"  # We tell the user there was an error.

return render_template('index.html', temperature=temperature, description=description, city=city_name)  # Show the webpage with the temperature
```

HTML Code that links with the Flask App

```html
<body>
    <h1>Weather in {{ city }}</h1>
    <!-- Displays the city name from the Flask app -->

    <div class="weather-box">
        <!-- This box contains the weather details -->

        <p><strong>Temperature:</strong> {{ temperature }}°C</p>
        <!-- Displays the temperature from the API -->

        <p><strong>Condition:</strong> {{ description }}</p>
        <!-- Displays the weather condition (e.g., "Cloudy") -->
    </div>
</body>

</html>
```

Later, when I build and test the app, you'll see that the Python Flask and HTML code work as expected with Jinja ({{ }}) replacing the placeholders with the weather data.

Building the flask app

Now the code and files were in order it was time to build the app and see if it was working.

1st Step: Installing required Python Packages

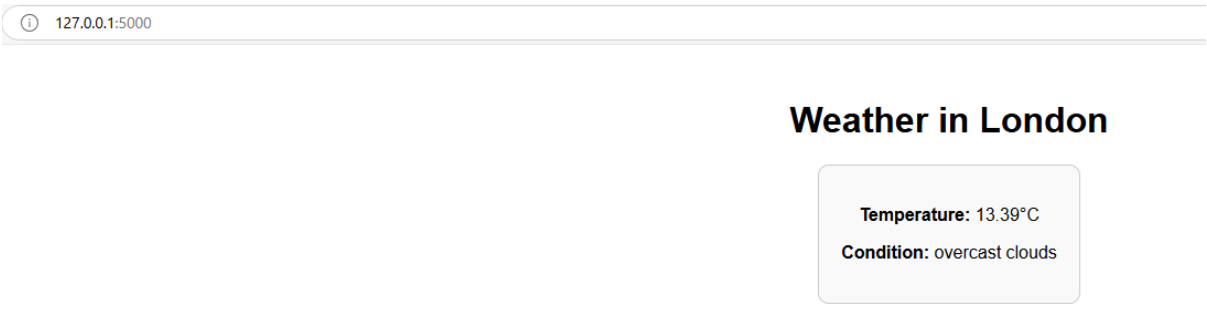
```
ModuleNotFoundError: No module named 'requests'
PS C:\Users\Harve\desktop\ECS> pip install requests
Collecting requests
  Downloading requests-2.32.3-py3-none-any.whl.metadata (4.6 kB)
Collecting charset-normalizer<4,>=2 (from requests)
  Downloading charset_normalizer-3.4.1-cp313-cp313-win_amd64.whl.metadata (36 kB)
Collecting idna<4,>=2.5 (from requests)
  Downloading idna-3.10-py3-none-any.whl.metadata (10 kB)
Collecting urllib3<3,>=1.21.1 (from requests)
  Downloading urllib3-2.3.0-py3-none-any.whl.metadata (6.5 kB)
Collecting certifi>=2017.4.17 (from requests)
  Downloading certifi-2025.1.31-py3-none-any.whl.metadata (2.5 kB)
Downloading requests-2.32.3-py3-none-any.whl (64 kB)
Downloading certifi-2025.1.31-py3-none-any.whl (166 kB)
Downloading charset_normalizer-3.4.1-cp313-cp313-win_amd64.whl (102 kB)
Downloading idna-3.10-py3-none-any.whl (70 kB)
Downloading urllib3-2.3.0-py3-none-any.whl (128 kB)
Installing collected packages: urllib3, idna, charset-normalizer, certifi, requests
Successfully installed certifi-2025.1.31 charset-normalizer-3.4.1 idna-3.10 requests-2.32.3 urllib3-2.3.0

[notice] A new release of pip is available: 24.3.1 -> 25.0.1
[notice] To update, run: python.exe -m pip install --upgrade pip
PS C:\Users\Harve\desktop\ECS>
```

2nd Step: Running the app locally by executing the python script to start the flask web server

Ran by using the following PS command: python app.py

3rd Step: Checking the app was running on port 5000

Using http://127.0.0.1:5000 (Localhost)



Confirmed now that the app is running locally, the data is being retrieved correctly, and the HTML/CSS styles are applied as expected, ensuring the app's interface is visually consistent.

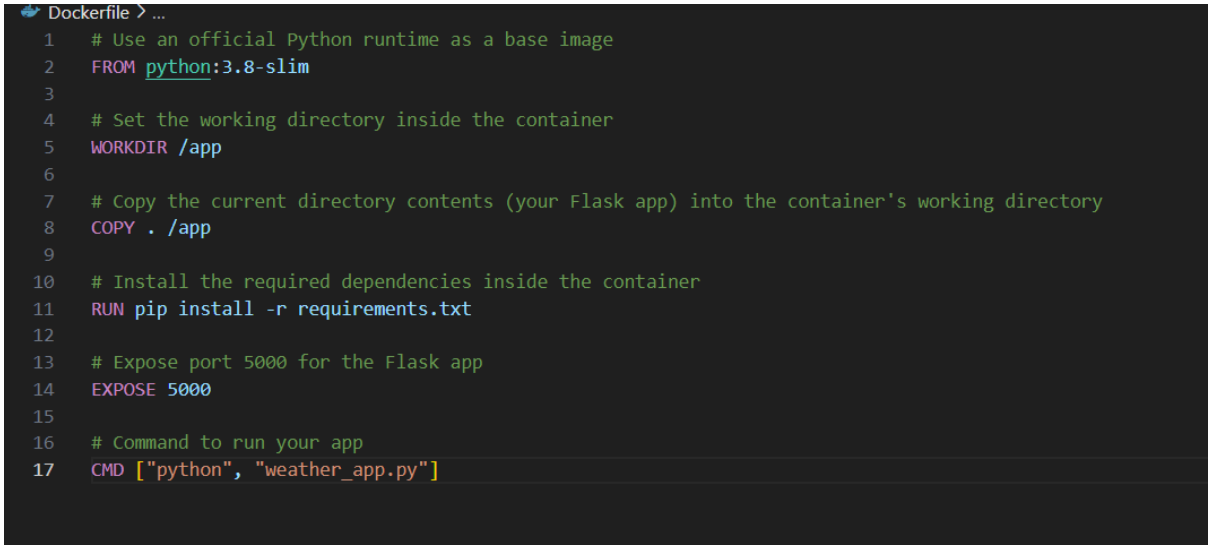
Containerize the Flask App using Docker

After a lengthy process of installing docker I was finally ready to start the main part of this project. The first step in doing so was to create a DockerFile.

What is a DockerFile?

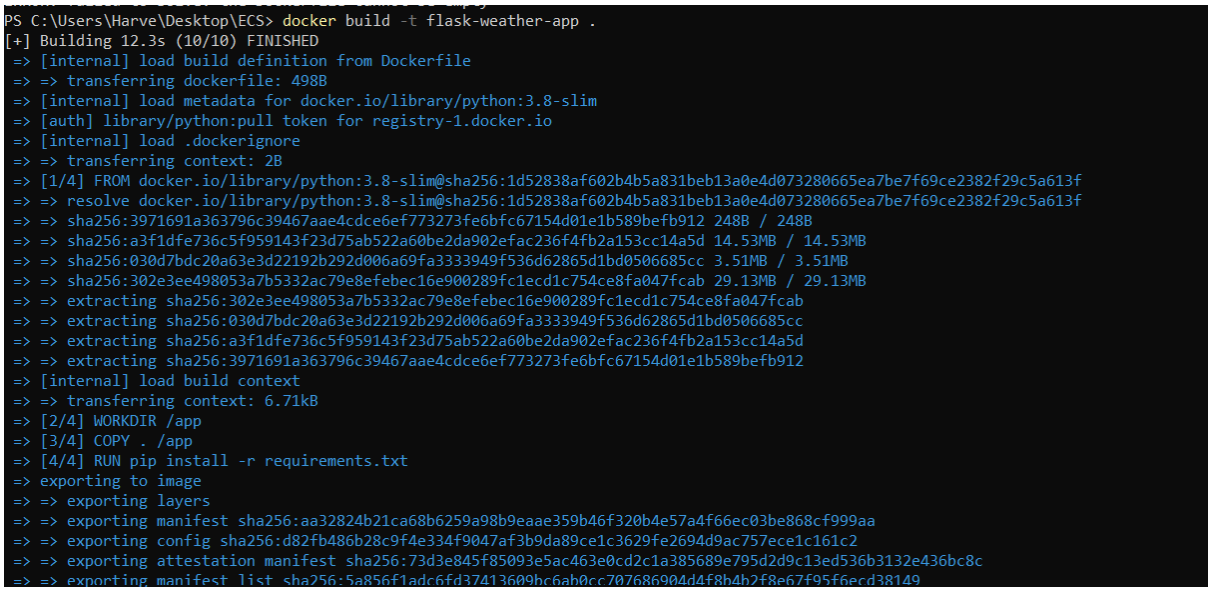
- A **Dockerfile** is a text file script that contains instructions for Docker to build a container image for an application.
- It defines the base image, copies necessary files, installs dependencies, and specifies the command to run the app inside the container.
- For example, a Dockerfile for a Flask app installs Flask, copies files, and runs the app on port 5000.

After researching and watching Docker tutorials, I created a simple Dockerfile to define the environment and instruct Docker to build the base image. This file specifies the base image, copies the necessary application files, installs dependencies, and sets the command to run the app inside the container.



Building the Docker container

After creating the Dockerfile, I ran the docker build -t flask-weather-app . command in PowerShell to build the Docker image. This command instructed Docker to process the Dockerfile and create an image named flask-weather-app, which could then be used to run containers.



Running the Flask App inside the container

I used the docker run command to start the container and the docker ps command to verify that it was running. This confirmed that the application was successfully deployed inside the Docker container.

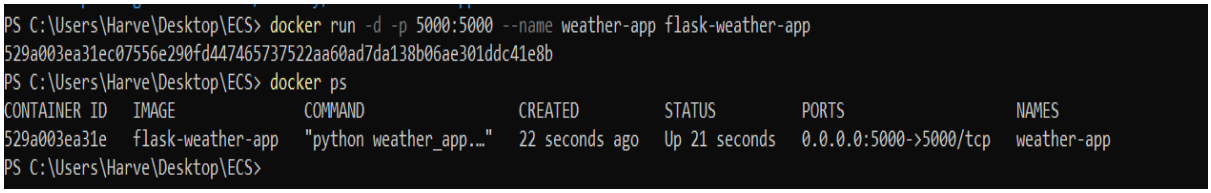
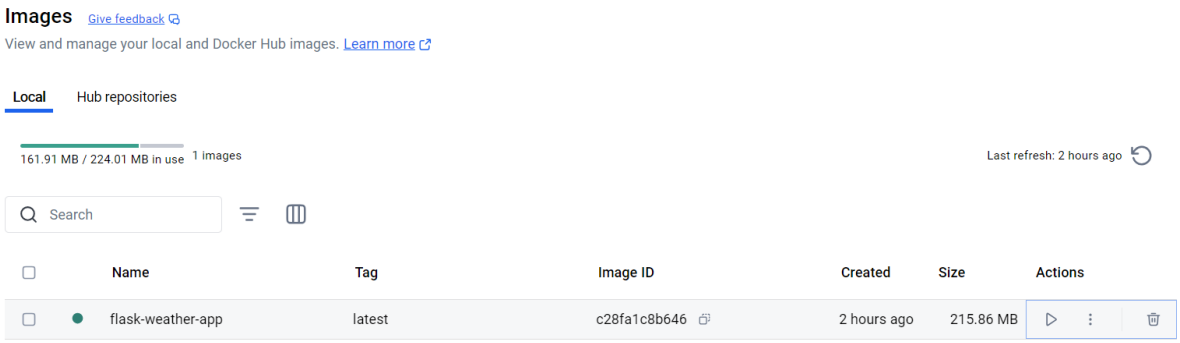
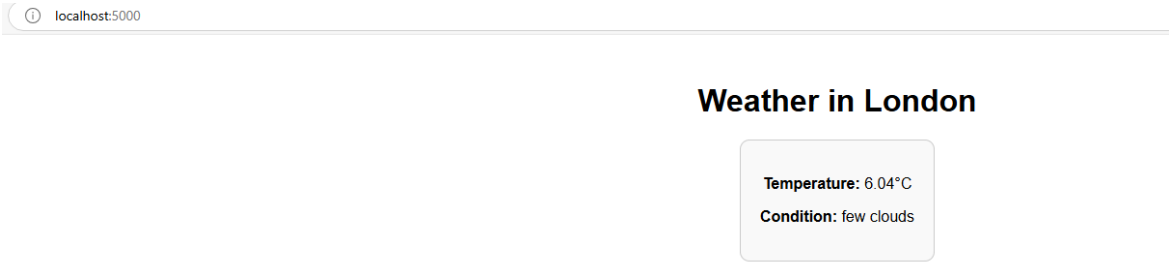


Image in Docker Desktop



Verifying app is running inside container

Once I verified that the container was running and the image was created, I checked the URL again to confirm that the app was successfully running inside the container and functioning as expected.



AWS Infrastructure deployed and managed via Terraform IAC

After successfully containerizing the Flask app with Docker, I used Terraform to define and deploy the necessary cloud infrastructure. The Terraform code automated the provisioning of resources.

AWS Infrastructure Breakdown

1. Networking

- Amazon VPC (Virtual Private Cloud):
    - A virtual network (aws_vpc) is created with a CIDR block of 10.0.0.0/16.
- Subnets:
    - Public and private subnets are created across two availability zones (eu-west-2a and eu-west-2b).
- Internet Gateway (aws_internet_gateway):
    - Attached to the VPC to enable internet access for public subnets.
- Route Tables:
    - Public and private route tables are configured to manage traffic routing.
- NAT Gateway (aws_nat_gateway):
    - Deployed in a public subnet to allow private subnets to access the internet securely.
- Elastic IP (EIP) (aws_eip):
    - Allocated and associated with the NAT Gateway to provide a public IP for outbound internet access.

2. Security

- Security Groups (aws_security_group):
    - ALB Security Group:
        - Allows inbound HTTP (port 80) and HTTPS (port 443) traffic.
        - Allows all outbound traffic.
    - ECS Security Group:
        - Restricts inbound traffic to only allow communication from the ALB on port 5000.
        - Allows all outbound traffic.

3. Load Balancing

- Application Load Balancer (ALB) (aws_lb):
    - Public-facing ALB deployed in public subnets.
    - Listens on port 80 (aws_lb_listener) and forwards traffic to the ECS tasks.
- Target Group (aws_lb_target_group):
    - Routes traffic to ECS tasks running on port 5000.
    - ALB performs health checks on the ECS tasks to ensure they are healthy and able to handle traffic.

4. Container Services

- Amazon ECS (Elastic Container Service):
    - ECS Cluster (aws_ecs_cluster):
        - Logical grouping of ECS tasks and services.
    - ECS Task Definition (aws_ecs_task_definition):
        - Defines the container image, CPU, memory, and networking requirements.
        - Uses AWS Fargate for serverless container execution.
    - ECS Service (aws_ecs_service):
        - Manages the desired number of ECS tasks (2 in this case).
        - Deploys tasks in private subnets with no public IPs.
        - Associates tasks with the ALB target group.

5. Container Registry

- Amazon ECR (Elastic Container Registry) (aws_ecr_repository):
    - Private Docker image repository for storing the application container image.
    - Image scanning is enabled to detect vulnerabilities on push.

6. IAM (Identity and Access Management)

- IAM Roles:
    - ECS Task Execution Role:
        - Grants permissions to pull images from ECR and write logs to CloudWatch.
    - ECS Task Role:
        - Grants permissions for ECS tasks to access other AWS services (e.g., S3, CloudWatch).
    - ECR Push Role:
        - Grants permissions to push Docker images to the ECR repository.
- IAM Policies:
    - Custom policies are attached to roles to define specific permissions (e.g., pushing images to ECR).
    -

I have included an un-modularized version of the terraform code which has been annotated for the sake of reviewing, please see attached file in the GIT repository.

Pushing the Docker Image to ECR

After defining and deploying the TF code I checked to see if ECS and ECR were created and running in the console, after verifying I then pushed the Docker Image to ECR.

The whole process of pushing the Docker image initially stumped me, as each use case can be different. I came across information online suggesting that shell scripts are often used to automate the Docker login process, because the login session typically expires after 12 hours. This means that if you need to make any updates to the app, you will have to go through the entire login process again.

However, since my app was intended as a one-time deployment for the purpose of showcasing my working experience, I decided to manually push the image. The infrastructure was destroyed after the project was completed, so it didn't make sense to automate a process for an app that would only be deployed once. The need for

automating the login and push process was unnecessary in this case, as I would not be frequently pushing new images or updating the app.

Logging in to docker via AWS CLI

```
PS C:\Windows\system32> aws ecr get-login-password --region eu-west-2 | docker login --username AWS --password-stdin [redacted].dkr.ecr.eu-west-2.amazonaws.com
Login Succeeded
```

Following the commands instructed in the AWS console I pushed the image to ECR

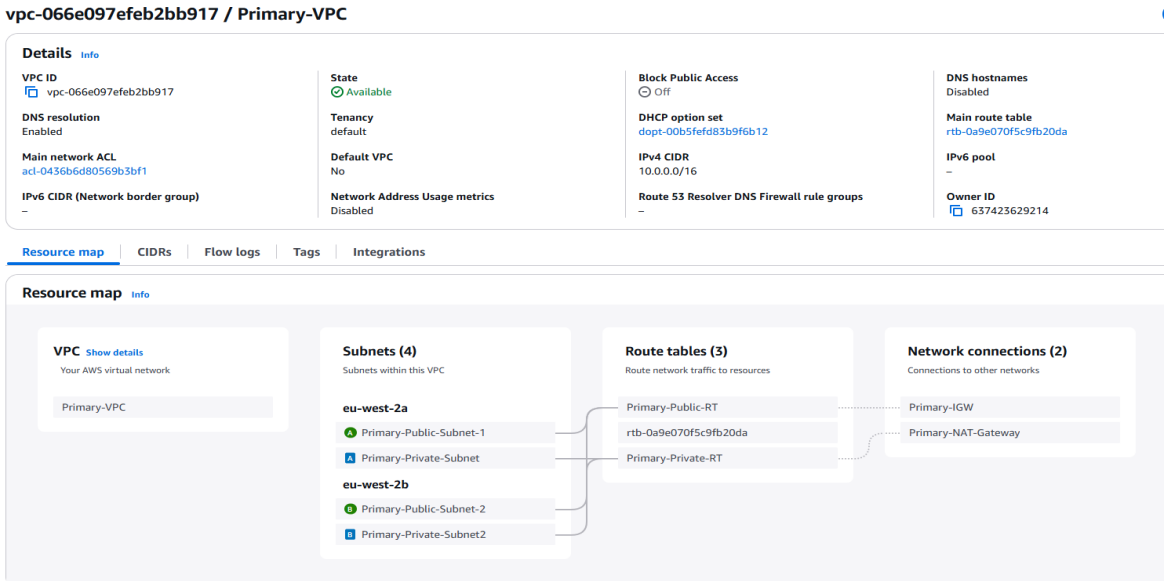
```
PS C:\Windows\system32> docker tag flask-weather-app:latest [redacted].dkr.ecr.eu-west-2.amazonaws.com/weather-repo:latest
PS C:\Windows\system32> docker push [redacted].dkr.ecr.eu-west-2.amazonaws.com/weather-repo:latest
The push refers to repository [[redacted].dkr.ecr.eu-west-2.amazonaws.com/weather-repo]
a3f1dfe736c5: Pushed
6897804f8b41: Pushed
3971691a3637: Pushed
302e3ee49805: Pushed
030d7bdc20a6: Pushed
9eec49d4cddc: Pushed
dc76dc40244a: Pushed
db8a87431f0b: Pushed
latest: digest: sha256:c28fa1c8b646da8135d42618adadc6a02cc838ebfce3041d319d07da8e3d1b01 size: 856
PS C:\Windows\system32>
```

Deployment checks in AWS console

After pushing the docker image to ECR the only task left to do for this project was to do a final test of the app and see if all the infrastructure had been created as expected.

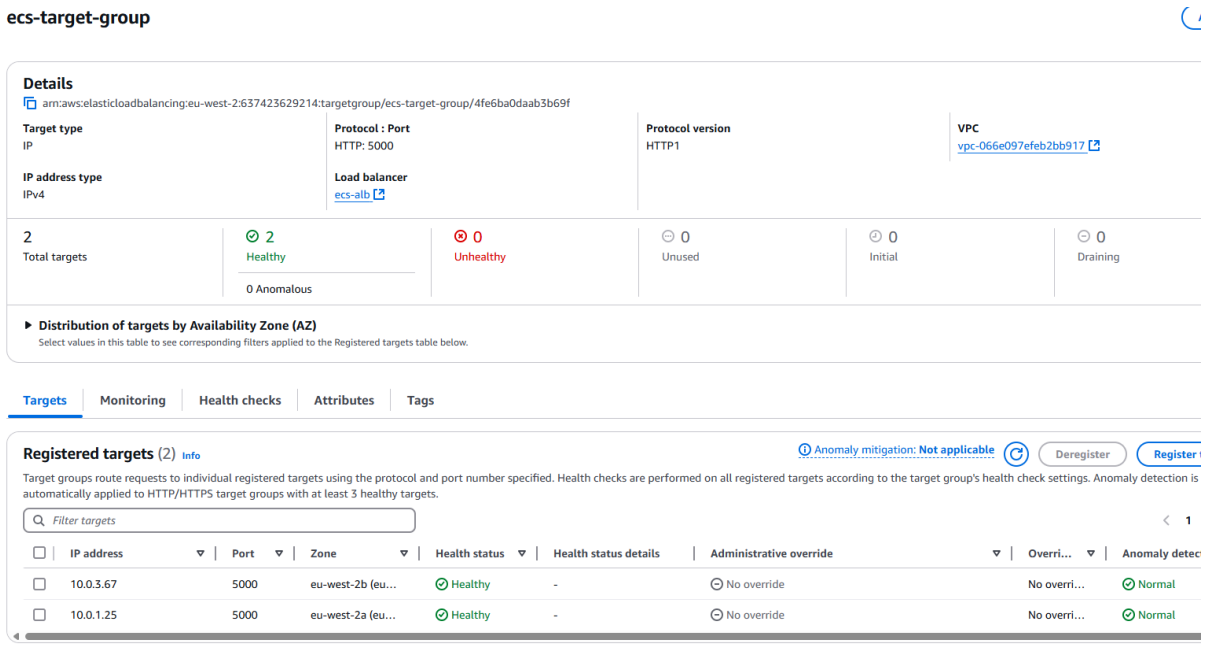
Networking

Here we have the **VPC** map showcasing the network topology and how the resources link together.



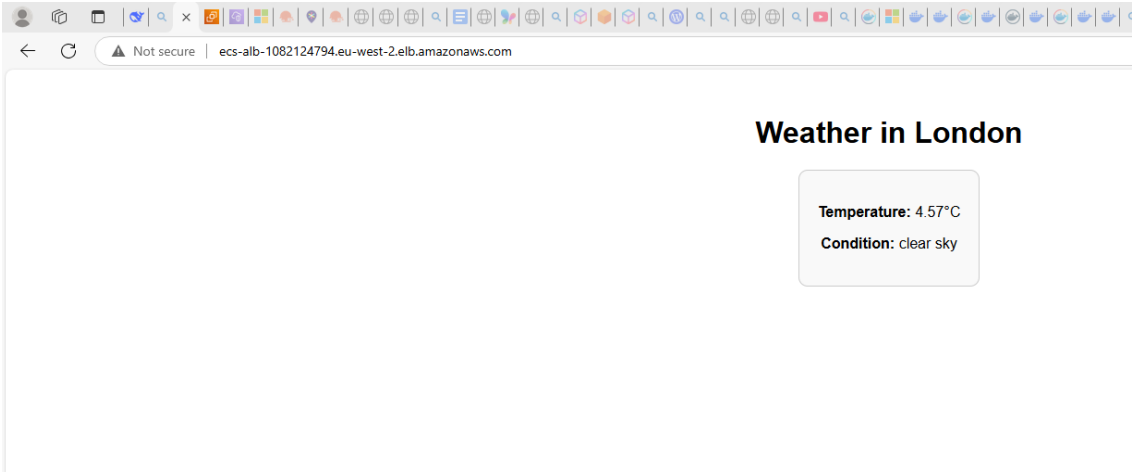
Load Balancing

Confirming the ALB listens on port 80 for HTTP traffic and routes it to ECS tasks running on port 5000 via a target group. The target group performs health checks on the ECS tasks to ensure only healthy tasks receive traffic.

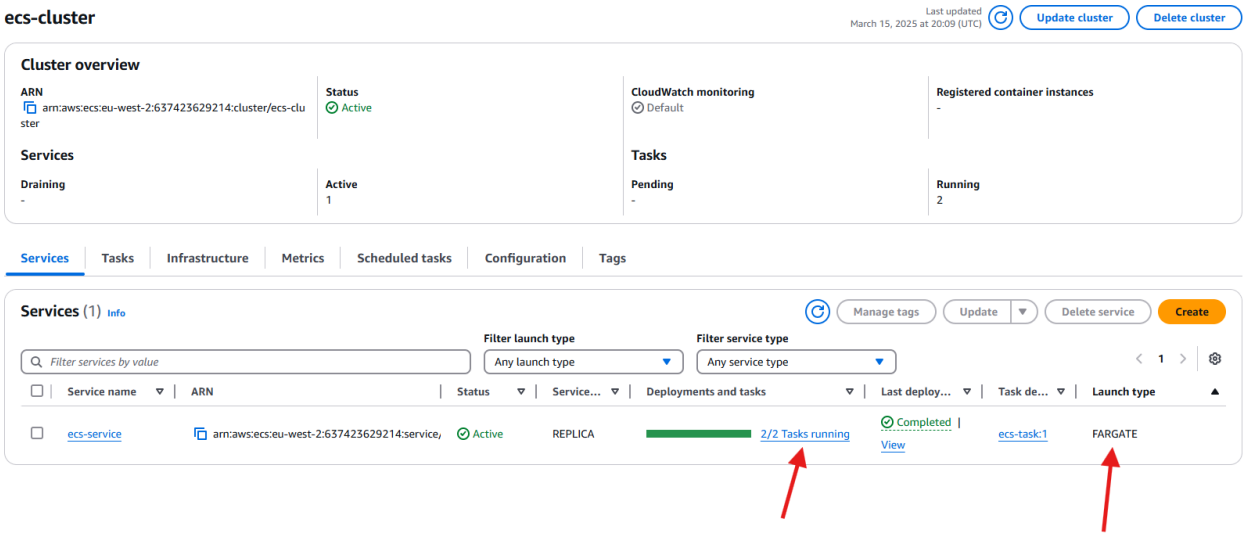


Checking ALB DNS

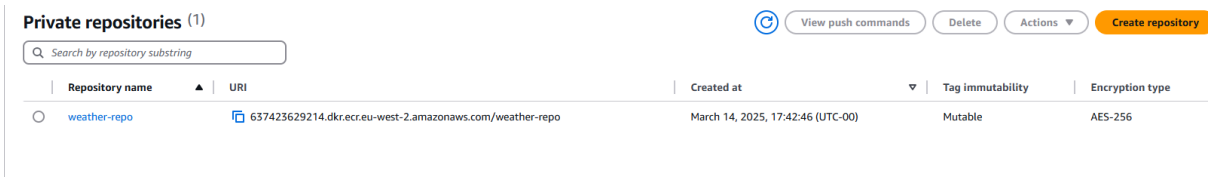
To confirm the ALB is functioning correctly and balancing the load for the application, the ALB DNS name is used for verification. Since this project does not have a real domain name, the ALB DNS name acts as a substitute to validate that the ALB is routing traffic as expected.



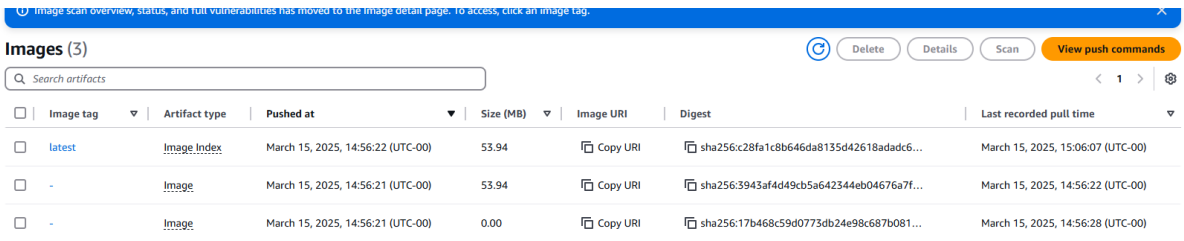
ECS Cluster, ECS tasks provisioned & FARGATE Launch Type.



ECR Repo



Confirming Docker image has been pushed to ECR Repo



Health check set up for target group to use ECS running tasks

In the original deployment of this infrastructure, health checks were not configured for the ECS tasks, meaning the load balancer had no way to verify the health or availability of the tasks. Leveraging the skills I gained from earning the AWS Solutions Architect Associate (SAA) certification, I implemented health checks to add redundancy and improve reliability. This ensures the load balancer routes traffic only to healthy tasks, enhancing the overall resilience of the application.

Adding in the ALB Target Group Health Check

Before

```
# Target Group for ECS Service
resource "aws_lb_target_group" "alb_target_group" {
  name        = "ecs-target-group"
  port        = 80
  protocol    = "HTTP"
  target_type = "ip"  # For Fargate, use IP target type

  vpc_id = aws_vpc.ecs_vpc.id
}
```

After

```
# Target Group for ECS Service
resource "aws_lb_target_group" "alb_target_group" {
  name        = "ecs-target-group"
  port        = 5000
  protocol    = "HTTP"
  target_type = "ip"  # For Fargate, use IP target type
  vpc_id = aws_vpc.Primary_vpc.id

  health_check {
    protocol            = "HTTP"
    path                = "/health"  # Health check URL for ECS tasks
    interval            = 30
    timeout             = 5
    healthy_threshold   = 3
    unhealthy_threshold = 3
  }

  tags = {
    Name        = "ECS Target Group"
    Environment = "production"
  }
}
```

The health check ensures ECS tasks are healthy by sending HTTP requests to the **/health** endpoint every 30 seconds. A task is marked healthy after 3 successful responses (200 OK) or unhealthy after 3 failures. The target group port is 5000 because the ECS tasks run the application on this port, ensuring traffic is forwarded to the correct endpoint.

Adding in the /health route in the flask app

Once the ALB target group (TG) code was updated to include health checks, I added a /health route to the Flask app. This route allows the ALB to send HTTP requests to the /health endpoint and receive a 200 OK response when the application is healthy. This ensures the ALB can properly monitor the ECS tasks and route traffic only to those that are functioning correctly.

```python
#Health check
@app.route('/health')
def health():
    return "OK", 200  # ALB will receive a 200 status, meaning the app is healthy.
```

Modifying the ECS Task Definition (to reflect the health check path)

Exposes port 5000 where my Flask App (with /health endpoint) runs.

```
#ECS task Definition
resource "aws_ecs_task_definition" "ecs_task" {
  family                   = "ecs-task"
  requires_compatibilities = ["FARGATE"]
  cpu                       = "256"
  memory                    = "512"
  network_mode              = "awsvpc"

  container_definitions = jsonencode([{
    name      = "my-container"
    image     = "${aws_ecr_repository.weather-app-repo.repository_url}:latest"
    memory    = 512
    cpu       = 256
    essential = true
    portMappings = [{
      containerPort = 5000
      hostPort      = 5000
      protocol      = "tcp"
    }]
  }])
  execution_role_arn = aws_iam_role.ecs_task_execution_role.arn
}
```

ECS service integration

Binds the ALB to my containers port with automatic registration/deregistration of tasks.

```hcl
#ECS using Fargate
resource "aws_ecs_service" "ecs_service" {
  name            = "ecs-service"
  cluster         = aws_ecs_cluster.ecs_cluster.id
  task_definition = aws_ecs_task_definition.ecs_task.arn
  desired_count   = 2  # desired count of running tasks

  launch_type = "FARGATE"

  network_configuration {
    subnets          = [aws_subnet.Primary_sub_priv.id, aws_subnet.Primary_sub_priv2.id]
    assign_public_ip = false  # Fargate tasks don't require public IPs
    security_groups  = [aws_security_group.ecs_sg.id]
  }

  load_balancer {
    target_group_arn = aws_lb_target_group.alb_target_group.arn
    container_name   = "my-container"
    container_port   = 5000
  }

  depends_on = [
    aws_lb_listener.ecs_listener
  ]
}
```

Security group update to allow ALB to ECS communication over port 5000

```hcl
# Security Group for ECS Tasks (restricted to only allow traffic from ALB)
resource "aws_security_group" "ecs_sg" {
  name        = "ecs-sg"
  description = "Allow traffic from ALB only"
  vpc_id      = aws_vpc.Primary_vpc.id

  # Allow inbound traffic from the ALB security group on HTTP/HTTPS ports
  ingress {
    from_port   = 5000
    to_port     = 5000
    protocol    = "tcp"
    security_groups = [aws_security_group.alb_sg.id]  # Only allow traffic from ALB
  }



  # Allow outbound traffic (usually all traffic)
  egress {
    from_port   = 0
    to_port     = 0
    protocol    = "-1" # All traffic
    cidr_blocks = ["0.0.0.0/0"]
  }
}
```

The ECS security group only allows inbound traffic on port 5000 from the ALB's security group, ensuring tasks can't be accessed directly from the internet. Outbound traffic is unrestricted to let tasks pull images and send logs. This creates a secure one-way flow:

internet traffic must pass through the ALB first, protecting the containers while maintaining functionality.

Full Workflow:

1. The ALB sends health checks (GET /health) to each ECS task's private IP on port 5000.Your Flask app returns 200 OK response
2. ALB marks task healthy after 3 good responses
3. Unhealthy tasks get terminated by ECS
4. Service launches replacement tasks automatically

Considerations and improvements

HTTPS Listener

One aspect not implemented in this deployment but essential for a fully functioning, production-ready application is an HTTPS listener for the Application Load Balancer (ALB).

For this project, HTTP (port 80) was intentionally used instead of HTTPS to simplify development and testing. Since this is a learning exercise without production traffic or sensitive data, HTTP reduces complexity by:

1. Avoiding certificate management – No need to provision and validate ACM certificates.
2. Simplifying debugging – HTTP allows easier inspection of traffic and logs without encryption overhead.
3. Reducing costs – No expenses for SSL/TLS certificates or domain registration.

While HTTPS is critical for real-world application ensuring security, SEO benefits, and compliance with browser requirement. This project prioritizes core AWS infrastructure components, such as ALB, ECS, and health checks.

Future HTTPS Implementation

To implement HTTPS in a production environment, the following steps would be required:

1. Request an ACM certificate – Use AWS Certificate Manager (ACM) to generate an SSL/TLS certificate for the domain.

2. Add an HTTPS listener – Configure the ALB to listen on port 443 and associate it with the ACM certificate.
3. Redirect HTTP to HTTPS – Enforce secure traffic by setting up automatic redirection from HTTP to HTTPS.
4. The alb_sg Security group is already configured for HTTP & HTTPS, so no changes needed here.

Open Egress security Group Rules

1. ALB Security Group (alb_sg)
    - The ALB does not *initiate* outbound connections; it only receives incoming traffic and forwards it to ECS tasks.
    - Open egress (0.0.0.0/0) ensures that the ALB can send health check requests to backend services or interact with other AWS resources if needed (e.g., logging to CloudWatch, redirecting traffic).
2. ECS Security Group (ecs_sg)
    - The ECS tasks (containers) must be able to communicate externally for various reasons, such as:
        - Fetching dependencies or updates from the internet (if required).
        - Communicating with other AWS services (S3, DynamoDB, RDS, or third-party APIs).
        - Sending responses back to clients via the ALB.
    - If outbound traffic were restricted, the ECS tasks might not be able to access necessary services (e.g., an RDS database, external APIs, or even basic networking like DNS resolution).

NAT Gateway redundancy

My thought process was if both NAT Gateways are live and running, they will be incurring a lot of extra unnecessary cost which for me building this project is a main concern.

If I were to implement a dual NAT GATEWAY architecture, there would be two main points to consider. Cost saving and RPO recovery time.

How it could be done?

Having built a failover-based architecture in a previous project. I began thinking about how utilizing cloud watch alarms and lambda functions could achieve this NAT Gateway failover solution.

Cost-Saving Approach (Single NAT with Failover)

Using one active NAT Gateway with CloudWatch and Lambda failover minimizes costs by ensuring only one NAT processes traffic at a time. If it fails, CloudWatch triggers a Lambda function to update route tables, redirecting traffic to the backup NAT. This approach reduces hourly and data transfer costs while maintaining availability, making it ideal for budget-conscious environments.

High Availability Approach (Dual NAT with Static Routing)

A dual NAT setup with static routing ensures instant failover by distributing private subnets across two NATs. If one fails, the other continues handling traffic without delays. While this setup incurs higher costs, it eliminates downtime and is ideal for mission-critical workloads that require uninterrupted internet access.

I suppose it depends on what is more important for a company standpoint, I can see that the best probable solution would be to accept the extra cost of having two active NAT Gateways running and focus on providing the best possible service having minimum downtime which in turn saves cost and probably outweighs the cost of the two NAT Gateways. The golden question here would be how much is 2-3 minutes of your application being down going to cost you? It's a trade off to consider.

Project Summary

This project was a great experience for me in terms of building a wider range of working knowledge with non-AWS services like Docker. The idea of containerization was always interesting to me and having an opportunity to implement it into something I have built was rewarding, especially sitting back and watching it all come together during the testing period. This project has further motivated me to look outside the realm of AWS cloud computing and look into more diverse deployments in the future.

Through this project, I gained the confidence and practical experience needed to manage Docker deployments effectively. I also developed a deeper understanding of how Docker integrates with AWS services like ECS and ECR, and how the overall container workflow operates. This experience has equipped me with the skills and knowledge to tackle similar Docker-related projects confidently in the future.