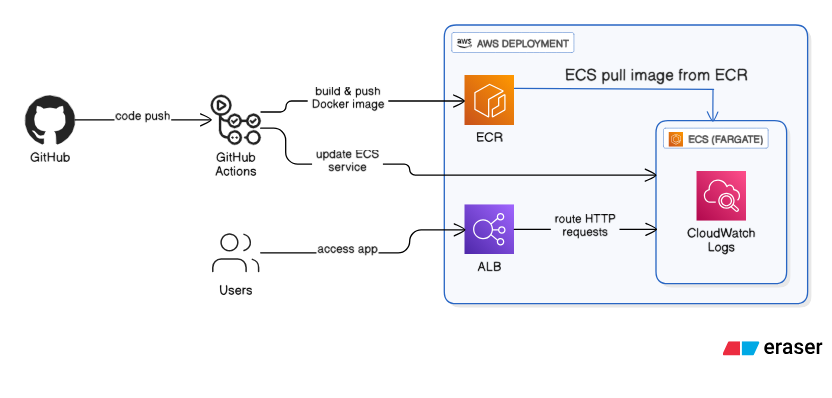
**End-to-End Cloud Deployment with AWS ECS, ECR, and GitHub Actions**

This guide will walk you through deploying a **Flask containerized app** on **AWS ECS (Fargate)** using **Amazon ECR** for storing Docker images and **Application Load Balancer (ALB)** for traffic distribution.  
We will do everything **from scratch** so even a beginner can follow along.

**Architectural Diagram:**

****

**🔹 Step 1: Prerequisites**

Before starting, make sure you have:

* An **AWS account** (free tier works)
* Basic understanding of **Docker** and **Flask app**
* AWS CLI installed (optional but recommended)
* Your app (example: app.py) ready

Example Flask app (app.py):

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def home():

return "Hello from ECS Fargate with ALB!"

@app.route('/health')

def health():

return "the container is healthy",200

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=5000)

**🔹 Step 2: Create Docker Image**

1. Create a Dockerfile in your project folder:
2. FROM python:3.9-slim
3. WORKDIR /app
4. COPY requirements.txt requirements.txt
5. RUN pip install -r requirements.txt
6. COPY . .
7. CMD ["python", "app.py"]
8. Build Docker image locally:
9. docker build -t ecs-flask-demo .
10. Test locally:
11. docker run -p 5000:5000 ecs-flask-demo

Visit: <http://localhost:5000>

**🔹 Step 3: Push Image to Amazon ECR**

Amazon ECR = **Elastic Container Registry** (private Docker repo on AWS).

1. Go to **AWS Console → ECR**
   * Click **Create repository**
   * Name it: ecs-flask-demo
   * Visibility: **Private**
   * Click **Create**
2. Authenticate Docker with ECR:
3. aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin <account\_id>.dkr.ecr.us-east-1.amazonaws.com
4. Tag and push image:
5. docker tag ecs-flask-demo:latest <account\_id>.dkr.ecr.us-east-1.amazonaws.com/ecs-flask-demo:latest
6. docker push <account\_id>.dkr.ecr.us-east-1.amazonaws.com/ecs-flask-demo:latest

✅ Now your image is stored in **ECR**.

**🔹 Step 4: Create ECS Cluster**

Amazon ECS = Elastic Container Service (we use **Fargate** for serverless containers).

1. Go to **AWS Console → ECS**
2. Click **Create Cluster**
3. Select **Networking only (Fargate)**
4. Name cluster: ECSClusterDemo
5. Click **Create**

✅ Cluster is ready.

**🔹 Step 5: Create Task Definition**

ECS Task = Blueprint of container settings.

1. Go to **ECS → Task Definitions → Create new task definition**
2. Launch type: **Fargate**
3. Task definition name: ECSTaskDemo
4. Task role: None (for now)
5. Add container:
   * Name: ecs-container
   * Image URI: (ECR image URI you copied earlier)
   * Port mappings: **5000**
6. CPU: **256 (.25 vCPU)**  
   Memory: **512 MiB**
7. Click **Create**

A screenshot of a computer

AI-generated content may be incorrect.

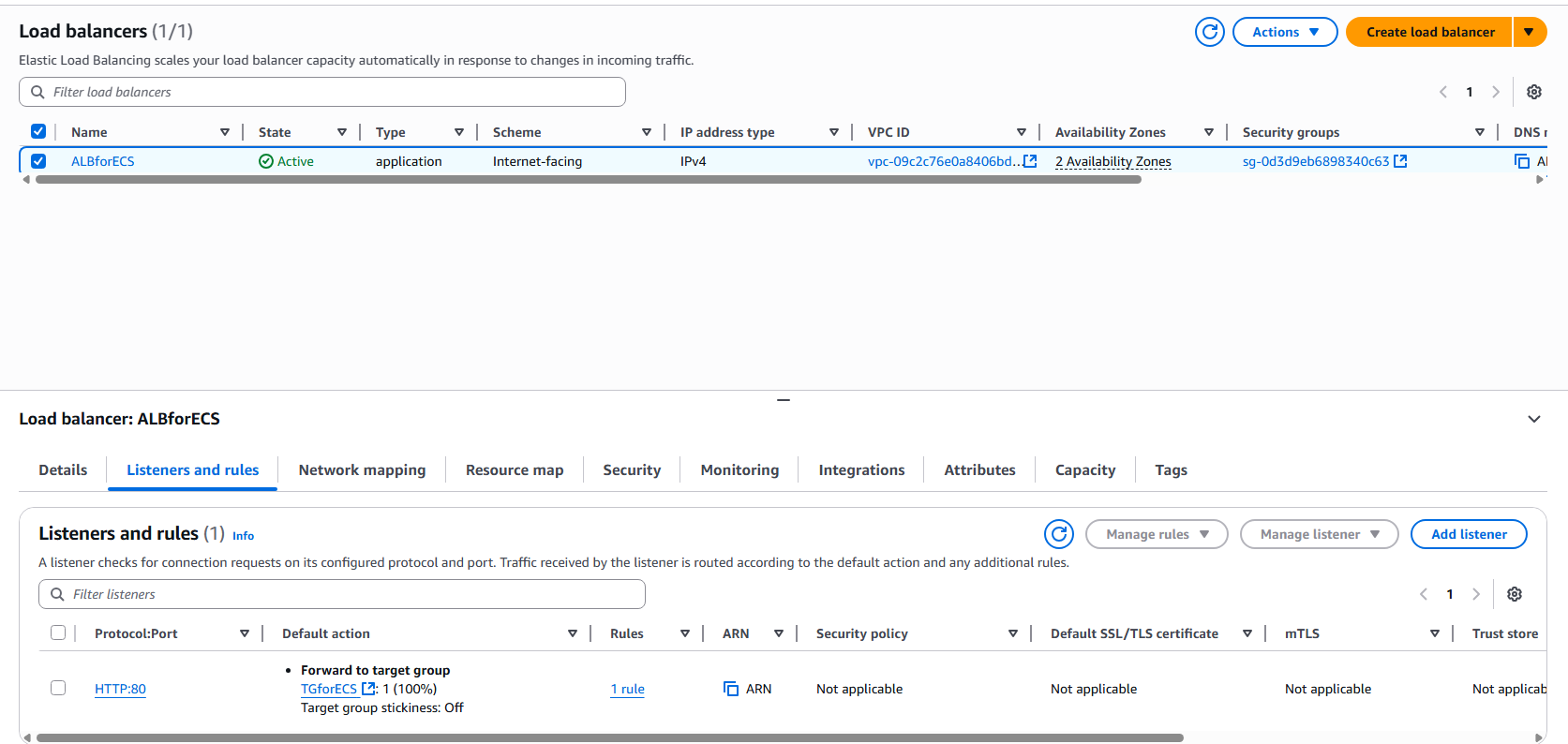
✅ Task definition created.

**🔹 Step 6: Create Application Load Balancer (ALB)**

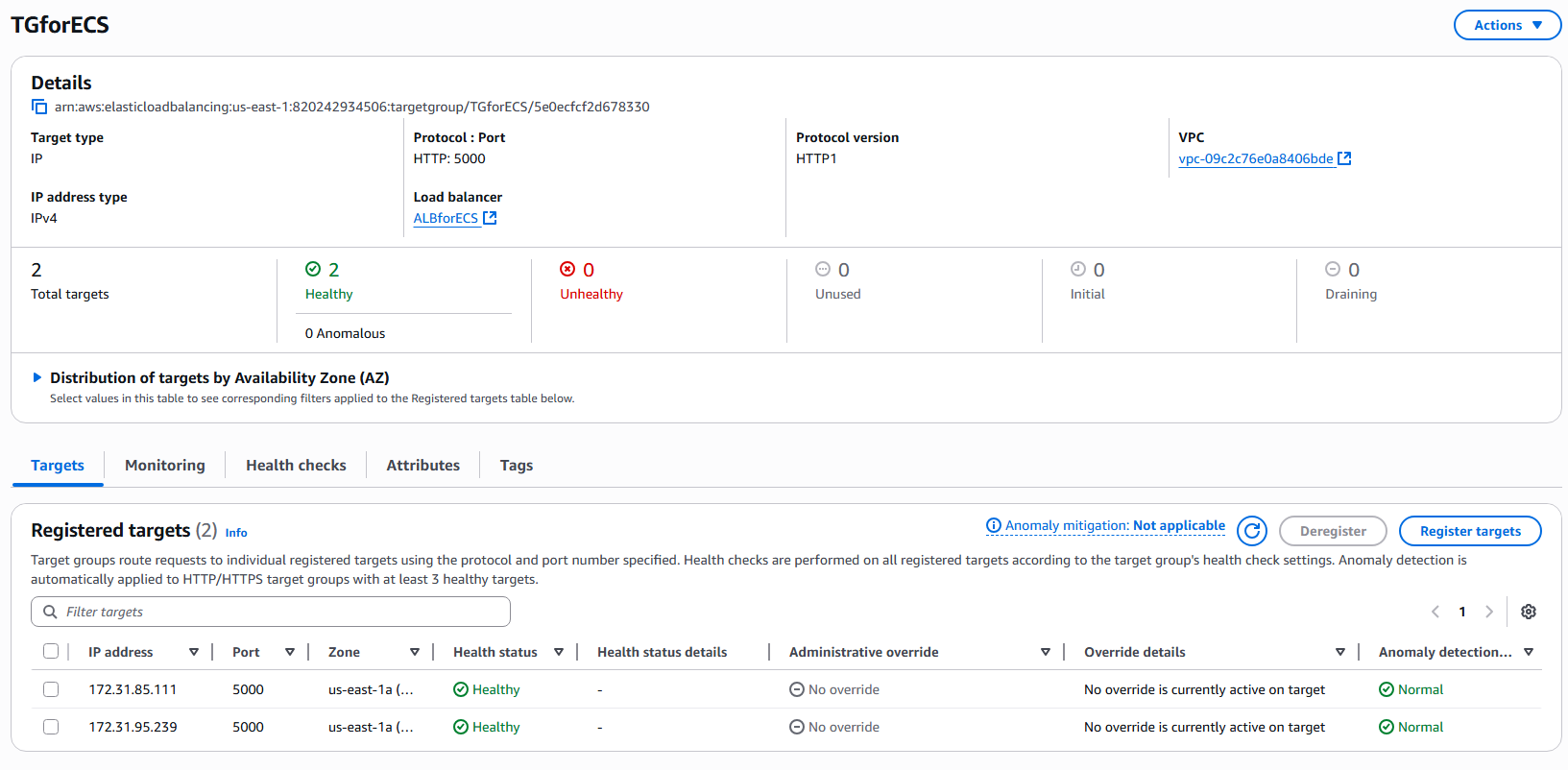
ALB = Distributes traffic to containers.

1. Go to **EC2 → Load Balancers → Create Load Balancer**
2. Select **Application Load Balancer**
3. Name: ALBforECS
4. Scheme: **Internet-facing**
5. IP type: IPv4
6. VPC: select default
7. Availability Zones: select at least **2 subnets**
8. Listeners:
   * HTTP :80 → Target group
9. Target Group:
   * Type: **IP**
   * Name: TGforECS
   * Protocol: HTTP
   * Port: **5000**
   * Health check path: / (important!)
10. Create Load Balancer.

**LOAD BALANCER**



**TARGET GROUP**

****

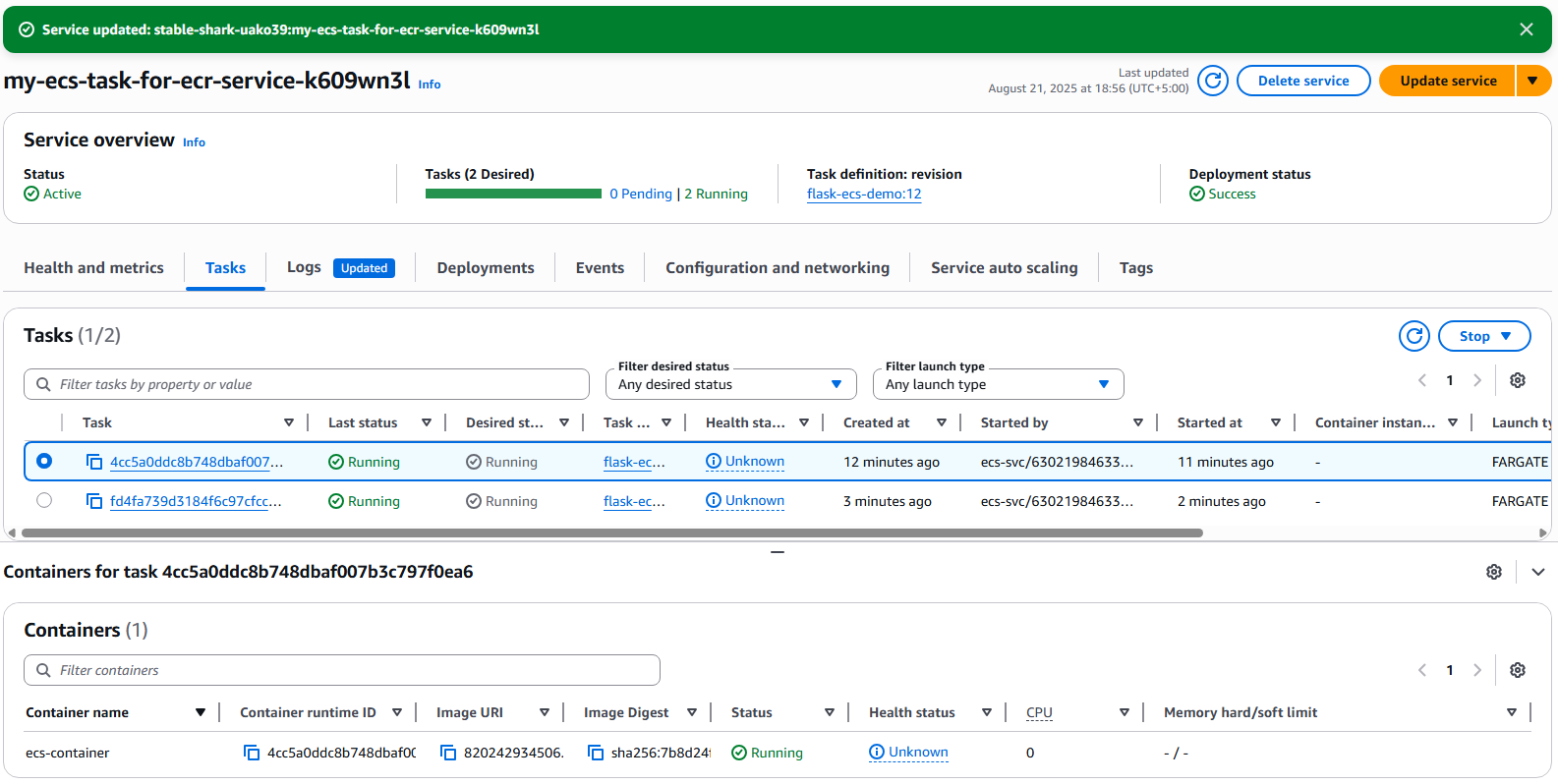
✅ ALB is ready with **Target Group TGforECS**.

**🔹 Step 7: Create ECS Service with ALB**

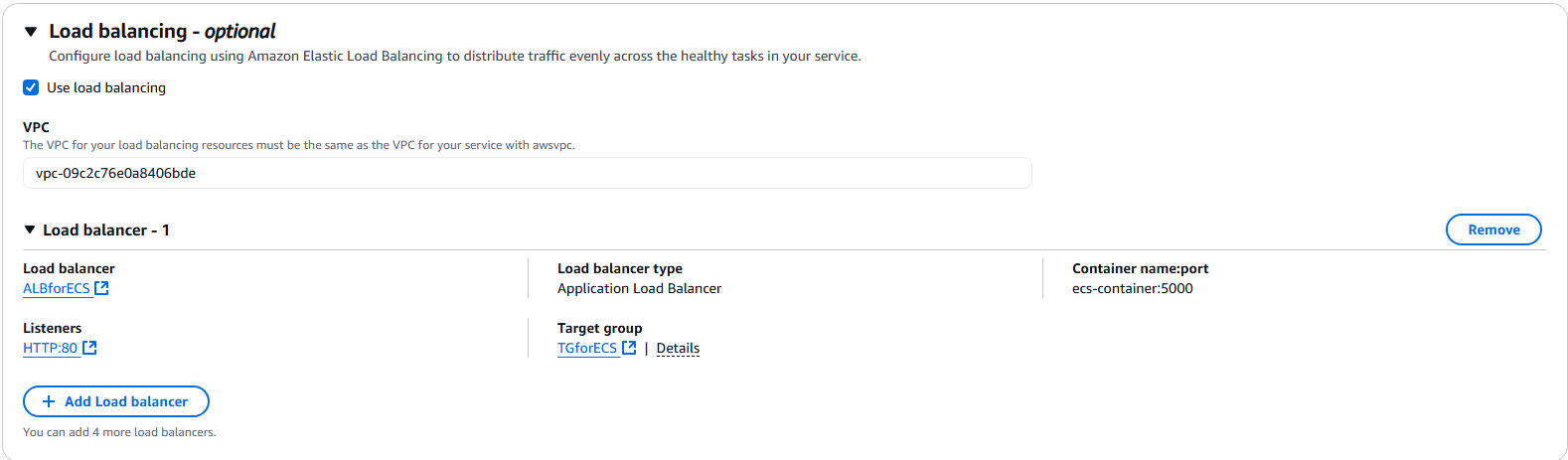
Now deploy containers inside ECS and attach ALB.

1. Go to **ECS → Clusters → ECSClusterDemo**
2. Click **Create Service**
3. Launch type: **Fargate**
4. Task definition: ECSTaskDemo
5. Service name: ECSServiceDemo
6. Desired tasks: 2 (for scaling)
7. Networking:
   * VPC: Default
   * Subnets: Pick 2
   * Security Group: Allow **HTTP (80) inbound**
8. Load balancing:
   * Select **Application Load Balancer**
   * Listener: HTTP:80
   * Target group: TGforECS
9. Click **Create Service**

✅ ECS service is running with **2 tasks**.

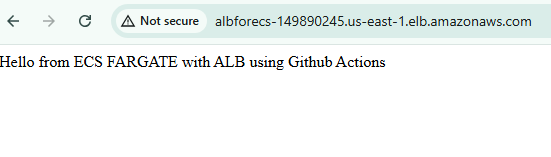


Load balancing setup with ECS.



**🔹 Step 8: Verify Deployment**

1. Go to **ECS → Cluster → ECSServiceDemo → Tasks**  
   ✅ Check that tasks show **RUNNING**.
2. Go to **EC2 → Target Groups → TGforECS → Targets**  
   ✅ Should show **Healthy**.  
   If **Unhealthy** → check:
   * Security group allows port 5000
   * Health check path is /
   * Container is listening on 0.0.0.0:5000
3. Copy **ALB DNS name** (from EC2 → Load Balancer → Description).
   * Open in browser:
   * http://<ALB-DNS-Name>
   * You should see:  
     **Hello from ECS Fargate with ALB and Github actions! Or your prompt**



**For /health**

A screenshot of a computer

AI-generated content may be incorrect.

**🔹 Step 9: Auto Scaling (Optional)**

1. Go to ECS Service → Auto Scaling
2. Add scaling policy:
   * Scale out if CPU > 70%
   * Scale in if CPU < 30%
3. Test by increasing load.

**🔹 Step 10: Cleanup (Important to avoid charges)**

When done:

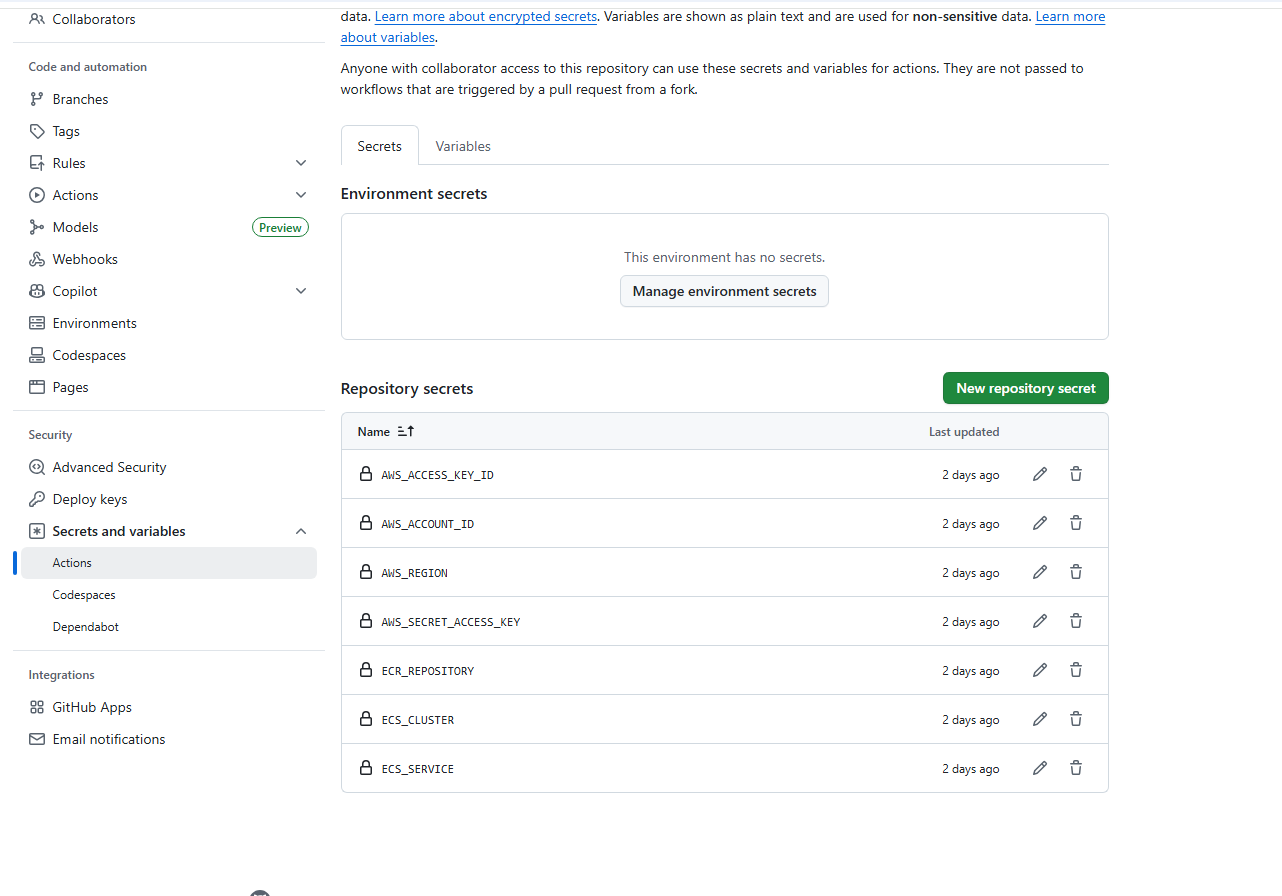
* Delete ECS Service
* Delete Cluster
* Delete ALB + Target Group
* Delete ECR repository
* Delete Security Groups if created manually

**Step 11: Automating Deployment with GitHub Actions**

Now let’s integrate **GitHub Actions** to automate the pipeline.

**📌 Step 11.1 – Create GitHub Repository**

1. Push your project (app.py, Dockerfile, requirements.txt) to GitHub
2. In GitHub → Go to **Settings → Secrets and variables → Actions**  
   Add these secrets:
   * AWS\_ACCESS\_KEY\_ID
   * AWS\_SECRET\_ACCESS\_KEY
   * AWS\_REGION → us-east-1
   * ECR\_REPOSITORY → flask-ecs-demo
   * AWS\_ACCOUNT\_ID



**📌 Step 11.2 – Create GitHub Actions Workflow**

In your repo, create:

.github/workflows/deploy.yml

**Paste this:**

name: CI/CD to ECS

on:

push:

branches: [ "master" ] # deploy on pushes to main

env:

AWS\_REGION: ${{ secrets.AWS\_REGION }}

AWS\_ACCOUNT\_ID: ${{ secrets.AWS\_ACCOUNT\_ID }}

ECR\_REPOSITORY: ${{ secrets.ECR\_REPOSITORY }}

ECS\_CLUSTER: ${{ secrets.ECS\_CLUSTER }}

ECS\_SERVICE: ${{ secrets.ECS\_SERVICE }}

IMAGE\_TAG: ${{ github.sha }}

BUILD\_PLATFORM: ${{ secrets.BUILD\_PLATFORM || 'linux/amd64' }}

jobs:

deploy:

runs-on: ubuntu-latest

steps:

- name: Checkout

uses: actions/checkout@v4

- name: Configure AWS Credentials

uses: aws-actions/configure-aws-credentials@v4

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@v2

- name: Set up Docker Buildx

uses: docker/setup-buildx-action@v3

- name: Build and Push image

run: |

REGISTRY="${AWS\_ACCOUNT\_ID}.dkr.ecr.${AWS\_REGION}.amazonaws.com"

IMAGE\_URI="${REGISTRY}/${ECR\_REPOSITORY}:${IMAGE\_TAG}"

LATEST\_URI="${REGISTRY}/${ECR\_REPOSITORY}:latest"

echo "Building for platform: ${BUILD\_PLATFORM}"

docker buildx build \

--platform "${BUILD\_PLATFORM}" \

--tag "${IMAGE\_URI}" \

--tag "${LATEST\_URI}" \

--push \

.

echo "IMAGE\_URI=${IMAGE\_URI}" >> $GITHUB\_ENV

- name: Render task definition with new image

id: render

uses: aws-actions/amazon-ecs-render-task-definition@v1

with:

task-definition: task-definition.json

container-name: ecs-container

image: ${{ env.IMAGE\_URI }}

- name: Deploy to ECS Service

uses: aws-actions/amazon-ecs-deploy-task-definition@v2

with:

task-definition: ${{ steps.render.outputs.task-definition }}

service: ${{ env.ECS\_SERVICE }}

cluster: ${{ env.ECS\_CLUSTER }}

wait-for-service-stability: true

A screenshot of a computer

AI-generated content may be incorrect.

**📌 Step 11.3 – Workflow Explanation**

* **On push to main branch** → Trigger pipeline
* **Checkout repo** → Pulls your app code
* **AWS credentials** → Authenticate GitHub with AWS
* **Login to ECR** → Enables Docker push
* **Build & Push Image** → Builds new Docker image and pushes to ECR
* **Deploy to ECS** → Forces ECS to pull the new image and redeploy service

**📌 Step 11.4 – Test Deployment**

1. Commit & push to main branch
2. GitHub Actions will run
3. ECS Service will redeploy with new Docker image
4. Open ALB DNS URL and confirm update 🎉

**Step: Using task-definition.json in GitHub Actions(Step 12)**

**Step 12: Using task-definition.json in GitHub Actions**

**Create the Task Definition JSON**

**1.** Inside this folder, create a file called:

**task-definition.json**

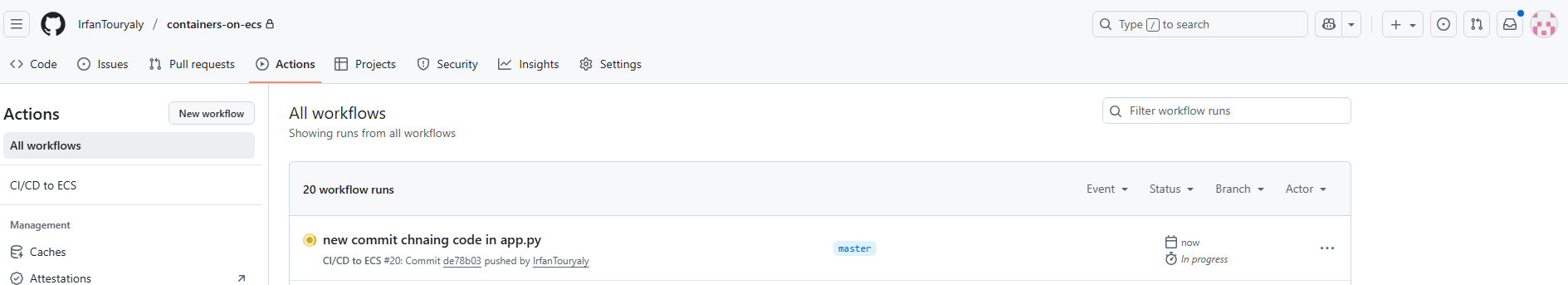
1. **Paste this code in Task Definition JSON inside:**

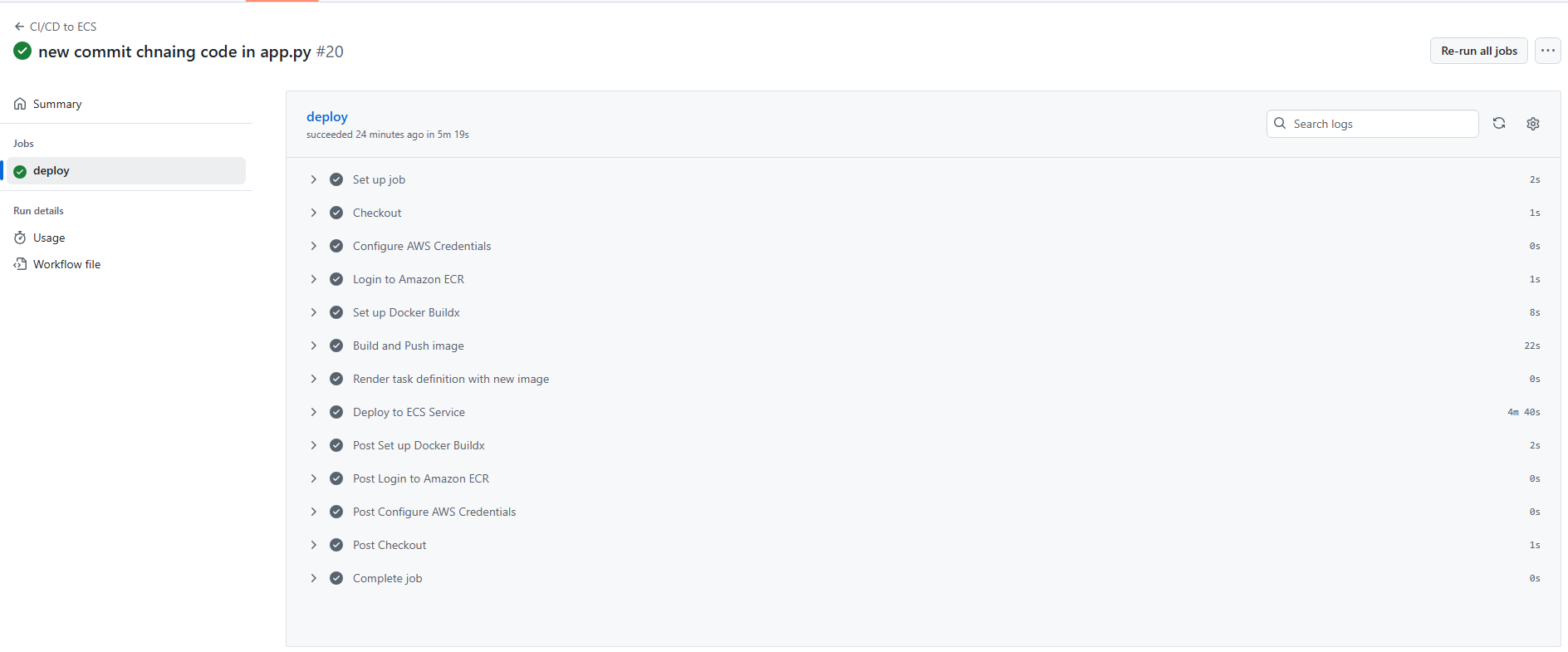
A screenshot of a computer

AI-generated content may be incorrect.

**Step 12.1 – Test Deployment**

1. Commit & push to main branch
2. GitHub Actions will run
3. ECS Service will redeploy with new Docker image
4. Open ALB DNS URL and confirm update 🎉





**✅ Final Outcome**

* Fully automated CI/CD pipeline
* Every push to GitHub main → builds Docker image → pushes to ECR → deploys ECS service with ALB