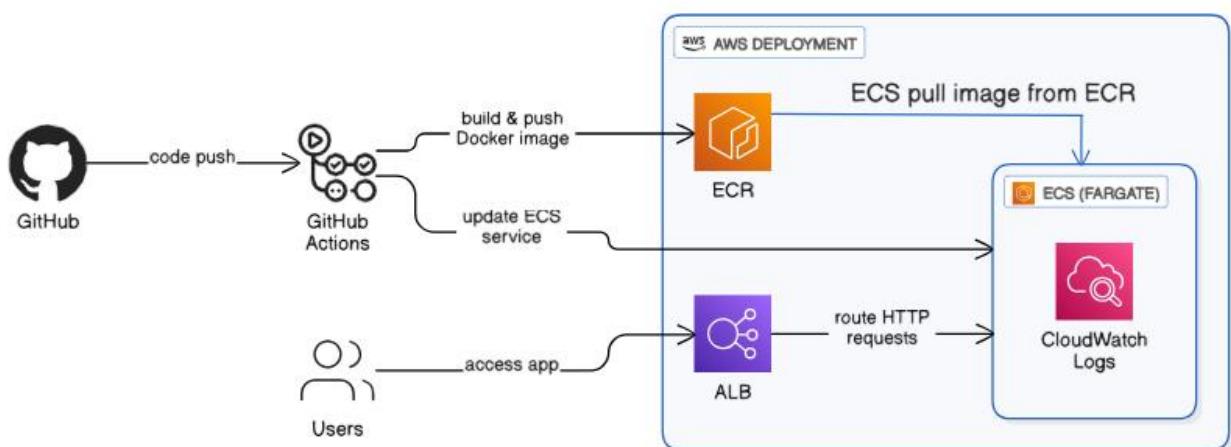


End-to-End Cloud Deployment with AWS ECS, ECR, and GitHub Actions

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**
 - o Click **Create repository**
 - o Name it: **ecs-flask-demo**
 - o Visibility: **Private**
 - o 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:
 - o Name: `ecs-container`
 - o Image URI: (ECR image URI you copied earlier)
 - o Port mappings: **5000**
6. CPU: **256 (.25 vCPU)**
Memory: **512 MiB**
7. Click **Create**

Revision	Status
flask-ecs-demo:12	ACTIVE
flask-ecs-demo:11	ACTIVE
flask-ecs-demo:10	ACTIVE
flask-ecs-demo:9	ACTIVE
flask-ecs-demo:8	ACTIVE
flask-ecs-demo:7	ACTIVE
flask-ecs-demo:6	ACTIVE
flask-ecs-demo:5	ACTIVE
flask-ecs-demo:4	ACTIVE
flask-ecs-demo:3	ACTIVE
flask-ecs-demo:2	ACTIVE
flask-ecs-demo:1	ACTIVE

✓ 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

The screenshot shows the AWS Load Balancers console. At the top, there is a search bar labeled "Filter load balancers" and a table with columns: Name, State, Type, Scheme, IP address type, VPC ID, Availability Zones, Security groups, and DNS name. One row is selected for "ALBforECS". Below this, a link "Load balancer: ALBforECS" leads to the detailed configuration page. This page has tabs for Details, Listeners and rules, Network mapping, Resource map, Security, Monitoring, Integrations, Attributes, Capacity, and Tags. The "Listeners and rules" tab is active, showing one rule for port 80 that forwards traffic to the "TGforECS" target group. A note says: "A listener checks for connection requests on its configured protocol and port. Traffic received by the listener is routed according to the default action and any additional rules."

TARGET GROUP

TGforECS

Details
[arn:aws:elasticloadbalancing:us-east-1:820242934506:targetgroup/TGforECS/5e0ecfcf2d678330](#)

Target type IP	Protocol : Port HTTP: 5000	Protocol version HTTP1	VPC vpc-09c2c76e0a8406bde		
IP address type IPv4	Load balancer ALBforECS				
2 Total targets	<input checked="" type="radio"/> 2 Healthy 0 Anomalous	<input type="radio"/> 0 Unhealthy	<input type="radio"/> 0 Unused	<input type="radio"/> 0 Initial	<input type="radio"/> 0 Draining

► **Distribution of targets by Availability Zone (AZ)**
Select values in this table to see corresponding filters applied to the Registered targets table below.

Targets | **Monitoring** | **Health checks** | **Attributes** | **Tags**

Registered targets (2) [Info](#)

Target groups route requests to individual registered targets using the protocol and port number specified. Health checks are performed on all registered targets according to the target group's health check settings. Anomaly detection is automatically applied to HTTP/HTTPS target groups with at least 3 healthy targets.

<input type="checkbox"/> Filter targets	<input type="checkbox"/> IP address	Port	Zone	Health status	Health status details	Administrative override	Override details	Anomaly detection...
<input type="checkbox"/>	172.31.85.111	5000	us-east-1a ...	<input checked="" type="radio"/> Healthy	-	<input type="radio"/> No override	No override is currently active on target	<input checked="" type="radio"/> Normal
<input type="checkbox"/>	172.31.95.239	5000	us-east-1a ...	<input checked="" type="radio"/> Healthy	-	<input type="radio"/> No override	No override is currently active on target	<input checked="" type="radio"/> Normal

[Anomaly mitigation: Not applicable](#) [Deregister](#) [Register targets](#)

✓ 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:
 - o VPC: Default
 - o Subnets: Pick 2
 - o Security Group: Allow **HTTP (80) inbound**
8. Load balancing:
 - o Select **Application Load Balancer**
 - o Listener: **HTTP:80**
 - o Target group: **TGforECS**
9. Click **Create Service**

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

The screenshot shows the AWS ECS Task Management interface for the service 'my-ecs-task-for-ecr-service-k609wn3l'. The 'Tasks' tab is selected, showing two tasks: one pending and two running. Both tasks are labeled 'Running' and have a status of 'Success'. The interface includes filters for task status, launch type, and creation time.

Load balancing setup with ECS.

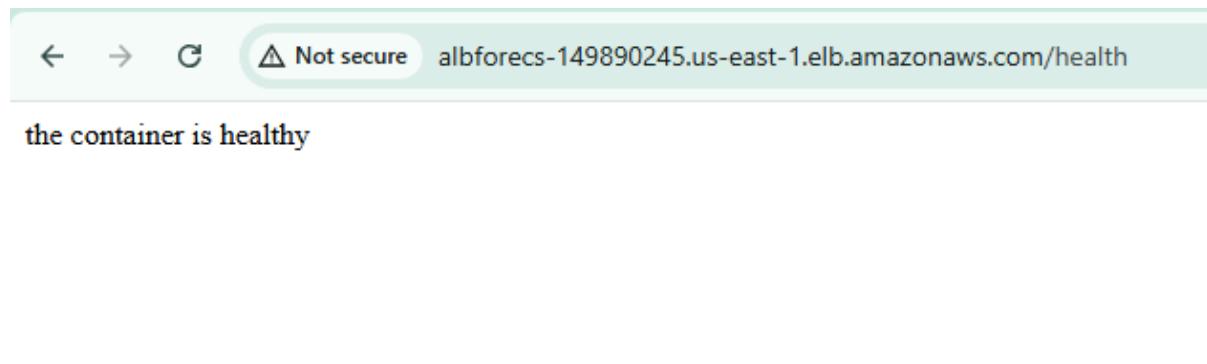
The screenshot shows the 'Load balancer - optional' section of the ECS service configuration. It includes options for using load balancing, selecting a VPC, and defining a load balancer. A single load balancer named 'ALBforECS' is configured with an Application Load Balancer type, listening on port 80, and targeting a target group 'TGforECS' which in turn targets the 'ecs-container' port 5000.

◆ 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



◆ 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

The screenshot shows the GitHub repository settings for managing secrets. On the left, there's a sidebar with various repository management options like Collaborators, Branches, Tags, Rules, Actions (which is currently selected), Models, Webhooks, Copilot, Environments, Codespaces, and Pages. Under Actions, there are sub-options for Codespaces and Dependabot. Below the sidebar, there are sections for Environment secrets and Repository secrets.

Environment secrets: A message says "This environment has no secrets." with a "Manage environment secrets" button.

Repository secrets: A table lists the following secrets:

Name	Last updated	Action
AWS_ACCESS_KEY_ID	2 days ago	edit delete
AWS_ACCOUNT_ID	2 days ago	edit delete
AWS_REGION	2 days ago	edit delete
AWS_SECRET_ACCESS_KEY	2 days ago	edit delete
ECR_REPOSITORY	2 days ago	edit delete
ECS_CLUSTER	2 days ago	edit delete
ECS_SERVICE	2 days ago	edit delete

A green "New repository secret" button is located at the top right of the Repository secrets section.

📌 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}" \  
--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

```

name: CI/CD to ECS
on:
  push:
    branches: [ "master" ] # deploy on pushes to main
  workflow_dispatch:
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout
        uses: actions/checkout@v2
      - name: Configure AWS Credentials
        uses: aws-actions/configure-aws-credentials@v4
        with:
          role_arn: ${{ secrets.AWS_ROLE_ARN }}
          access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          region: ${{ secrets.AWS_REGION }}
      - name: Login to ECR
        id: login-ecr
        uses: docker/login-action@v2
      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v1
      - name: Build and Push Image
        run:
          - echo "Building for platform: ${{ matrix.platform }}"
          docker build . --tag ${{ matrix.tag }}
          - platform: "${{ matrix.platform }}"
            - tag: "${{ matrix.tag }}_m1"
            - tag: "${{ matrix.tag }}_m2"
            - tag: "${{ matrix.tag }}_m3"
          - push
      - name: Deploy to ECS
        env:
          IMAGE_NAME: ${{ matrix.tag }}_m1
        uses: infraToursly/deploy-ecs@main

```

📌 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)

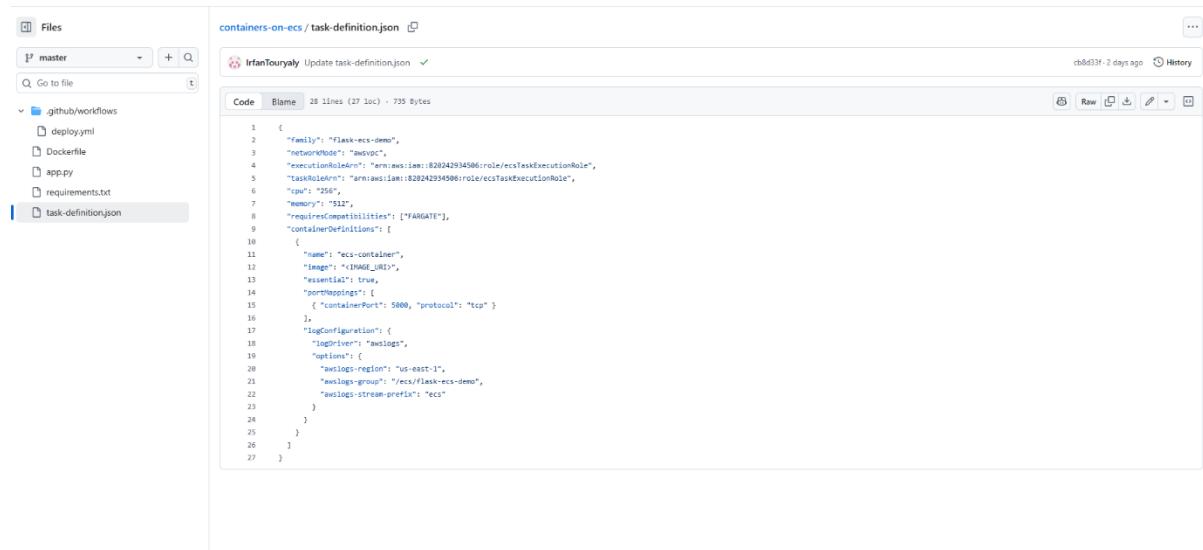
Step 12: Using task-definition.json in GitHub Actions

Create the Task Definition JSON

1. Inside this folder, create a file called:

task-definition.json

3. Paste this code in Task Definition JSON inside:



```
1  {
2    "family": "slack-ecs-demo",
3    "networkMode": "awsvpc",
4    "executionRoleArn": "arn:aws:iam::820242934596:role/ecsTaskExecutionRole",
5    "taskRoleArn": "arn:aws:iam::820242934596:role/ecsTaskExecutionRole",
6    "cpu": "256",
7    "memory": "512",
8    "requiresCompatibilities": ["FARGATE"],
9    "containerDefinitions": [
10      {
11        "name": "ecs-container",
12        "image": "iHAGD_URX",
13        "essential": true,
14        "portMappings": [
15          { "containerPort": 5000, "protocol": "tcp" }
16        ],
17        "logConfiguration": {
18          "logDriver": "awslogs",
19          "options": {
20            "awslogs-region": "us-east-1",
21            "awslogs-group": "/ecs/slack-ecs-demo",
22            "awslogs-stream-prefix": "ecs"
23          }
24        }
25      }
26    ]
27 }
```

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 🎉



The screenshot shows the GitHub Actions interface for the repository. On the left, there's a sidebar with navigation links: Actions (which is selected), New workflow, All workflows, CI/CD to ECS, Management, Caches, and Attestations. The main area displays the 'All workflows' section with a heading 'All workflow runs'. It says 'Showing runs from all workflows' and shows 20 workflow runs. A search bar at the top right says 'Filter workflow runs'. Below the search bar, there are dropdown menus for Event, Status, Branch, and Actor. One workflow run is visible in the list, titled 'new commit chnaing code in app.py', which triggered the 'CI/CD to ECS #20: Commit de78603 pushed by IrfanTouryal'. The status of this run is 'In progress' and it was started 'now'.

The screenshot shows the AWS CodePipeline console. At the top, there's a breadcrumb navigation: '← CI/CD to ECS' and a status message 'new commit chnaing code in app.py #20'. On the right, there are buttons for 'Re-run all jobs' and '...'. Below this, there's a sidebar with links: 'Summary', 'Jobs' (selected), 'Run details', 'Usage', and 'Workflow file'. The main area is titled 'deploy' and shows a successful run that completed 24 minutes ago in 5m 19s. It lists 14 steps: Set up job, Checkout, Configure AWS Credentials, Login to Amazon ECR, Set up Docker Buildx, Build and Push image, Render task definition with new image, Deploy to ECS Service, Post Set up Docker Buildx, Post Login to Amazon ECR, Post Configure AWS Credentials, Post Checkout, and Complete job. Each step has a timestamp next to it. A search bar for logs is at the top right.

✓ Final Outcome

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