

Deployment of Flask Backend & Express Frontend on AWS

Aim

The aim of this assignment is to deploy a complete full-stack application consisting of:

- **Flask backend (Python)**
- **Express frontend (Node.js)**

across three different deployment approaches on AWS:

1. Deployment on a **single EC2 instance**
2. Deployment on **two separate EC2 instances**
3. Deployment using **Docker containers with AWS ECR, ECS, and VPC**

The purpose of this assignment is to learn cloud deployment, containerization, AWS services, and real-world infrastructure setup practices.

1. Introduction

Modern applications often require separate frontend and backend components, deployed in flexible cloud environments. In this project, the Express frontend communicates with the Flask backend over REST APIs.

I deployed this system in three ways:

1. Traditional deployment on EC2
2. Distributed deployment (frontend + backend separated)
3. Containerized microservices deployment using ECR + ECS + VPC

This assignment demonstrates understanding of:

- Linux server configuration
 - EC2 setup & security groups
 - Docker containerization
 - Private container registry (ECR)
 - ECS cluster, task definitions & services
 - VPC networking concepts
-

2. Application Overview

2.1 Flask Backend (Python)

- Provides REST APIs.
- Handles business logic.
- Runs on port **5000**.
- Supports health-check endpoints.

2.2 Express Frontend (Node.js)

- Runs user interface.
- Sends API requests to Flask backend.
- Runs on port **3000**.

2.3 Communication Flow

Frontend → calls → Backend → returns → JSON response

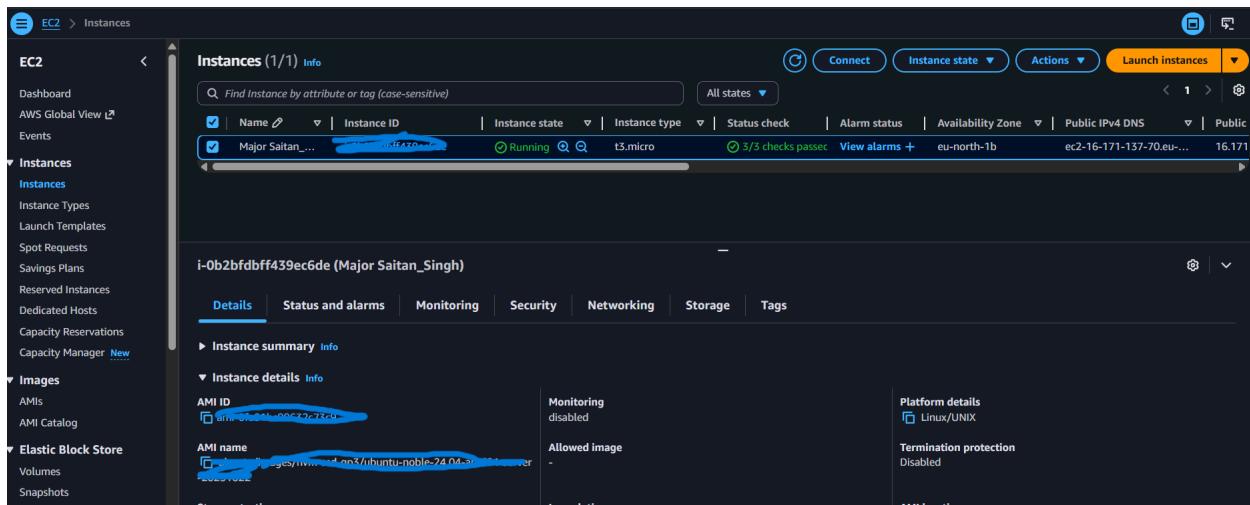
3. Deployment – Single EC2 Instance

In this method, both applications run inside a single EC2 instance.

Steps Performed

1. Created an EC2 instance (Ubuntu).
2. Installed:
 - o Node.js & npm
 - o Python3, pip
3. Cloned GitHub repository.
4. Started backend on port 5000 and frontend on 3000.
5. Modified security groups to allow both ports.

Output Screenshot



4. Deployment – Separate EC2 Instances

In this approach:

- One EC2 instance runs the **backend**
- One EC2 instance runs the **frontend**

Steps Performed

Backend EC2

1. Installed Python3 + pip
2. Ran Flask backend on port 5000
3. Allowed inbound rule: port **5000**

Frontend EC2

1. Installed Node.js
2. Updated API URL to point to backend instance's public IP
3. Allowed inbound rule: **3000**

Output Screenshot

Assignment 2 — Node (Frontend) → Flask (Backend)

Grade Checker

76

Get Grade

```
{ "result": "Grade: C" }
```

Student Grades (add/update)

Raju

B

Add / Update Student

Show All Students

```
{ "Raju": "B" }
```

```
{  
    "Raju": "B"  
}
```

Write to File

```
Hello raju, i hope you are doing well in your life
```

```
Write File
```

```
{  
    "result": "File written successfully."  
}
```

Read from File

```
Read File
```

```
{  
    "content": "Hello raju, i hope you are doing well in your life\n"  
}
```



Flask Backend Running — visit /health or POST to /submit

5. Deployment – Dockerized Using AWS ECR, ECS, VPC

This was the main part of the assignment and required detailed work.

5.1 Containerization Using Docker

Steps:

1. Created **Dockerfile for Flask backend**
2. Created **Dockerfile for Express frontend**
3. Built images using:

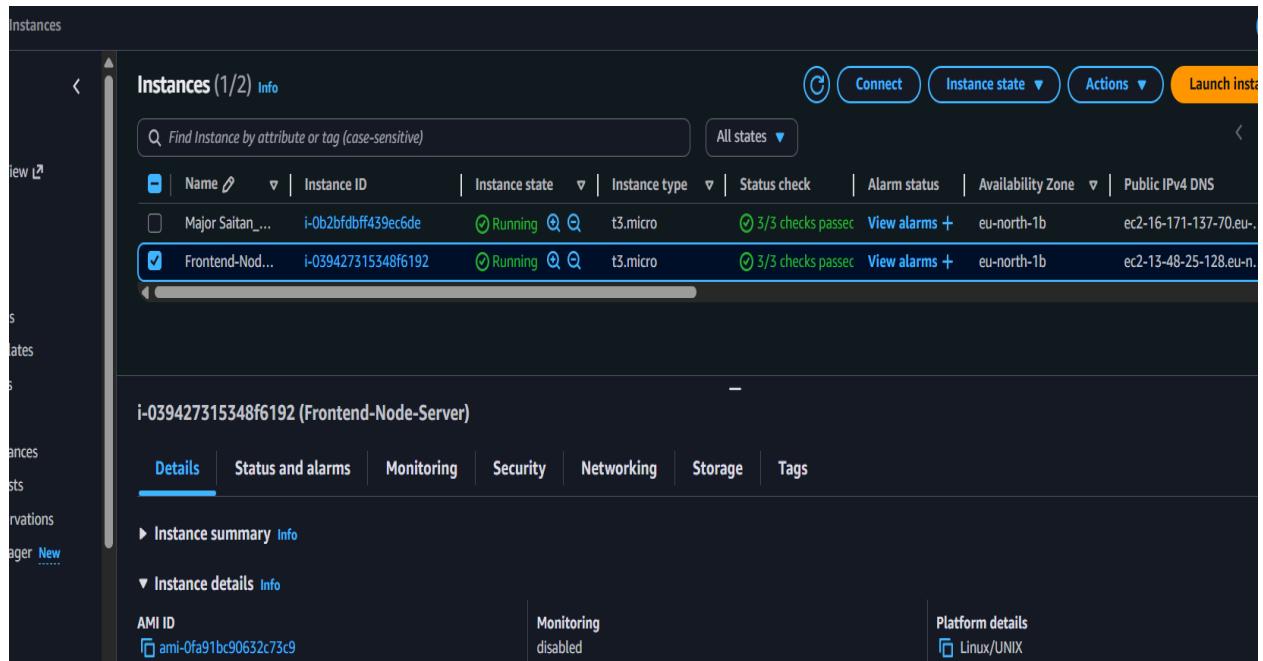
```
docker build -t flask-backend .
docker build -t node-frontend .
```

4. Tagging images:

```
docker tag flask-backend:latest
<aws-account>.dkr.ecr.<region>.amazonaws.com/flask-backend:latest
docker tag node-frontend:latest
<aws-account>.dkr.ecr.<region>.amazonaws.com/node-frontend:latest
```

5. Logged in to ECR and pushed images successfully.

Output Screenshot



The screenshot shows the AWS EC2 Instances page. At the top, there is a search bar and several filters: Name, Instance ID, Instance state, Instance type, Status check, Alarm status, Availability Zone, and Pub. The instance i-043ca1b60f9d41f98 is selected, and its details are shown below. The instance has three port mappings: 22 (TCP) to 0.0.0.0/0 via security group launch-wizard-3, 3000 (TCP) to 0.0.0.0/0 via security group launch-wizard-3, and 80 (TCP) to 0.0.0.0/0 via security group launch-wizard-3. Below the instance details, there is a section for Outbound rules, which lists a single rule allowing all traffic from 0.0.0.0/0 to all ports via security group launch-wizard-3.

5.2 AWS ECR (Elastic Container Registry)

- Created two private repositories:
 - **flask-backend**
 - **node-frontend**
- Pushed both container images from EC2 to ECR.

5.3 AWS ECS Cluster Creation

Steps:

1. Created a new **ECS Cluster** (Networking-only).
2. Created **Task Definitions** for:
 - **flask-backend-task**
 - **node-frontend-task**
3. Configured:
 - **Fargate launch type**

- CPU: .25 vCPU
- Memory: 0.5 GB
- Container ports (3000 and 5000)
- Execution role: **ecsTaskExecutionRole**

Output Screenshot

The image displays two screenshots of the AWS Elastic Container Service (ECS) console, illustrating the configuration of a cluster.

Screenshot 1: Cluster Configuration (Top)

This screenshot shows the "Clusters" section of the ECS console. It includes the following sections:

- Container Insights:** Describes Container Insights with enhanced observability as recommended. It provides detailed health and performance metrics at task and container level, along with aggregated metrics at cluster and service level. A note states: "Provides detailed health and performance metrics at task and container level in addition to aggregated metrics at cluster and service level. Enables easier drill downs for faster problem isolation and troubleshooting." The "Turned off" option is selected.
- ECS Exec encryption and logging:** Allows configuring logging and security settings for running commands in containers using ECS Exec. It includes fields for "Choose an AWS KMS key or enter an ARN" and a "Create an AWS KMS key" button.
- Logging for ECS Exec:** Offers three options: "Default" (selected), "Override", and "None". "Default" sends logs to CloudWatch Logs using the awslogs log driver. "Override" logs to a specified CloudWatch log group or Amazon S3 bucket. "None" means no logs are sent.
- Encryption - optional:** Provides options for "Managed storage" and "Fargate ephemeral storage".

Screenshot 2: Account Settings (Bottom)

This screenshot shows the "Account settings" section of the ECS console. It includes the following sections:

- CloudWatch Container Insights:** Shows "Turned off".
- Amazon ECS Runtime Monitoring:** Shows "Amazon ECS runtime monitoring" turned off.
- Resource Tagging Authorization:** Shows "Turned on".
- Default log driver mode:** Shows "Non-blocking".

> Clusters

Cluster flask-node-cluster has been created successfully.

Clusters (1) Info

Search clusters

Cluster	Services	Tasks	Container Instances
flask-node-cluster	0	No tasks running	0 EC2

5.4 ECS Service Creation

For each service (frontend + backend):

1. Selected cluster.
2. Created service using task definition.
3. Configured:
 - o VPC
 - o Subnets (public)
 - o Security group (port 3000 or 5000)
4. Enabled auto-assign public IP.

Output Screenshot

us-east-1.console.aws.amazon.com/ecs/v2/clusters/flask-node-cluster/services?region=us-east-1

Search [Alt+S] Account ID: 2393-7222-7948 United States (N. Virginia) Sultan_Singh

Container Service > Clusters > flask-node-cluster > Services

flask-backend-task-service-vr8ixir9 deployment is in progress. It takes a few minutes. View in CloudFormation X

Task definition successfully created flask-backend-task:1 has been successfully created. You can use this task definition to deploy a service or run a task. View task definition X

Cluster flask-node-cluster has been created successfully. View cluster X

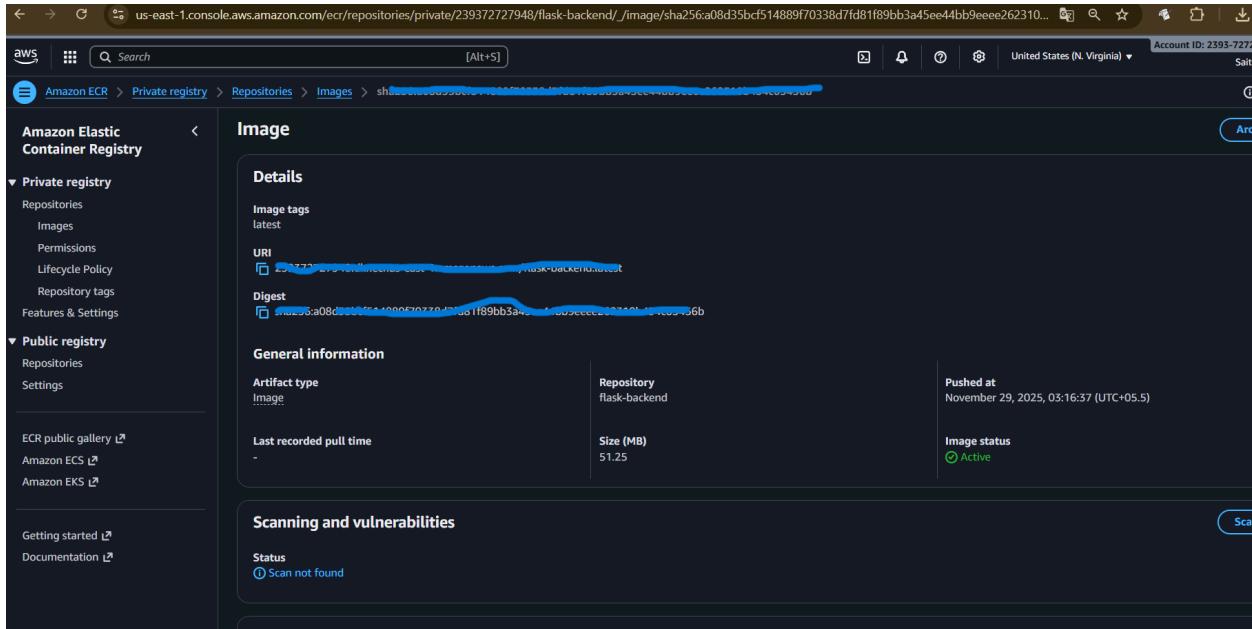
Notifications 0 0 0 2 0 0 1 Last updated

5.5 Successful Deployment

After debugging issues (image not found, role missing, ECR paths incorrect, etc.), the services were successfully deployed.

Both containers are now running inside ECS and reachable via public IP.

Output Screenshot



6. Issues Faced & Resolution

Issue 1: ECR repository not found

Cause: Wrong region in image push command

Fix: Created ECR in correct region & updated docker tag.

Issue 2: ECS service linked role missing

Fix: Deleted problematic role → recreated automatically.

Issue 3: Task stopped due to “CannotPullContainerError”

Fix: Corrected ECR URI & re-pushed image.

Issue 4: Frontend could not reach backend

Fix: Allowed backend's security group from anywhere OR from frontend service.

7. Final Outputs

Component	Status
Flask Backend	Deployed Successfully
Express Frontend	Deployed Successfully
EC2 (Single)	Working
EC2 (Separated)	Working
ECR	Images uploaded
ECS Cluster	Running
VPC Networking	Configured
Application	Live and accessible

8. Conclusion

This assignment gave hands-on experience with:

- Cloud deployment using AWS
- Running applications on EC2
- Designing distributed architecture
- Using Docker for containerization
- Pushing images to Amazon ECR
- Running containers using ECS Fargate
- Handling IAM roles, VPC, subnets, and security groups

By completing all three deployment methods, I learned the full workflow of:

Local development → Dockerization → Cloud deployment → Production hosting

This project strengthened my understanding of DevOps, AWS cloud infrastructure, CI/CD foundations, and container orchestration.