📑 Table of Contents

---

1. Introduction

This project demonstrates how to deploy a complete full-stack application on AWS using three different deployment strategies:

1. **Single EC2 instance deployment**
2. **Two EC2 instances deployment (frontend and backend separate)**
3. **Container deployment using Docker + AWS ECR + ECS + VPC**

The backend is built using **Flask (Python)** and the frontend using **Express.js (Node.js)**.

Skills gained include EC2 provisioning, security group management, Dockerization, ECS cluster operations, ECR usage, and VPC networking.

---

2. Application Architecture

2.1 Flask Backend

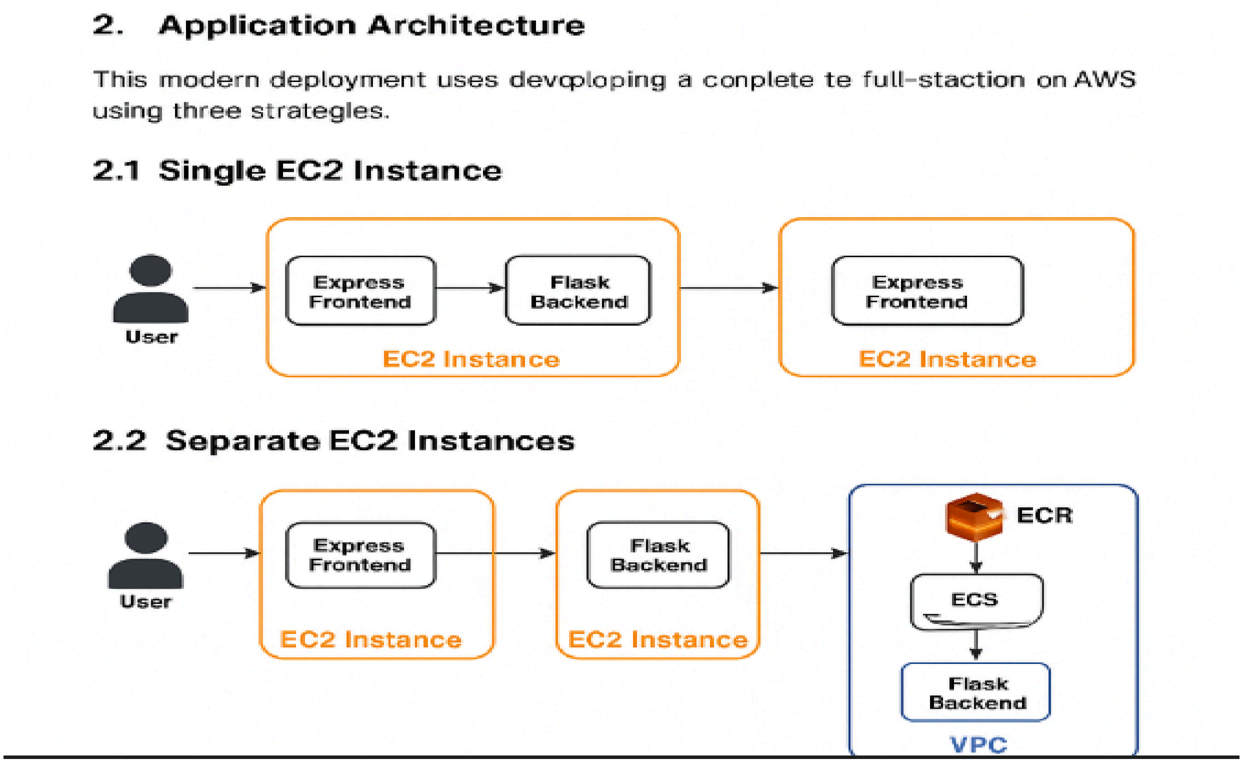
- Runs on **port 5000**
- Implements REST API
- Returns JSON responses

2.2 Express Frontend

- Runs on **port 3000**
- Sends requests to backend
- Displays results on UI

2.3 Overall Flow

```
User → Express Frontend → Flask Backend → JSON Response
```



2. Application Architecture

This modern deployment uses devqploping a conplete te full-staction on AWS using three strategles.

2.1 Single EC2 Instance

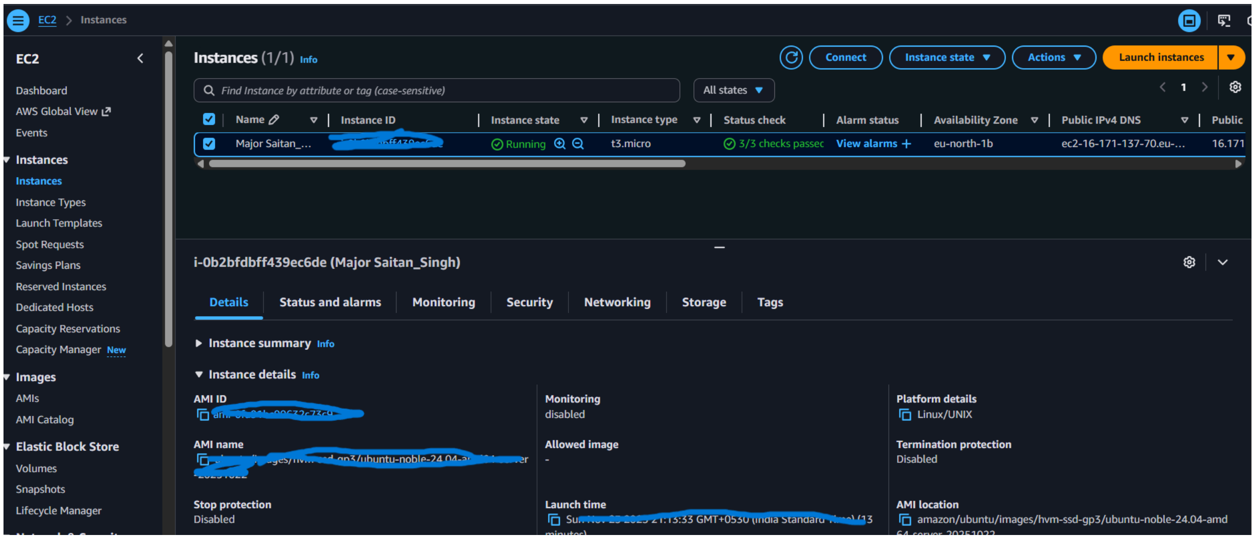
2.2 Separate EC2 Instances

3. Deployment on Single EC2 Instance

Both the frontend and backend were hosted on a **single Ubuntu EC2 instance** running in eu-north-1 region.

3.1 EC2 Instance Configuration

📷 Insert Image Here — EC2 Instance Running



Instances (1/1) Info

Find Instance by attribute or tag (case-sensitive)

☑	Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability Zone	Public IPv4 DNS	Public
☑	Major Saitan_...	[redacted]	Running	t3.micro	3/3 checks passec	View alarms +	eu-north-1b	ec2-16-171-137-70.eu-...	16.171

i-0b2bfdbff439ec6de (Major Saitan_Singh)

▶ Instance summary Info

▼ Instance details Info

AMI ID
[redacted]

Monitoring
disabled

Platform details
Linux/UNIX

AMI name
[redacted]

Allowed image
-

Termination protection
Disabled

Stop protection
Disabled

Launch time
[redacted]

AMI location
amazon/ubuntu/images/hvm-ssd-gp3/ubuntu-noble-24.04-amd64-server-20251022

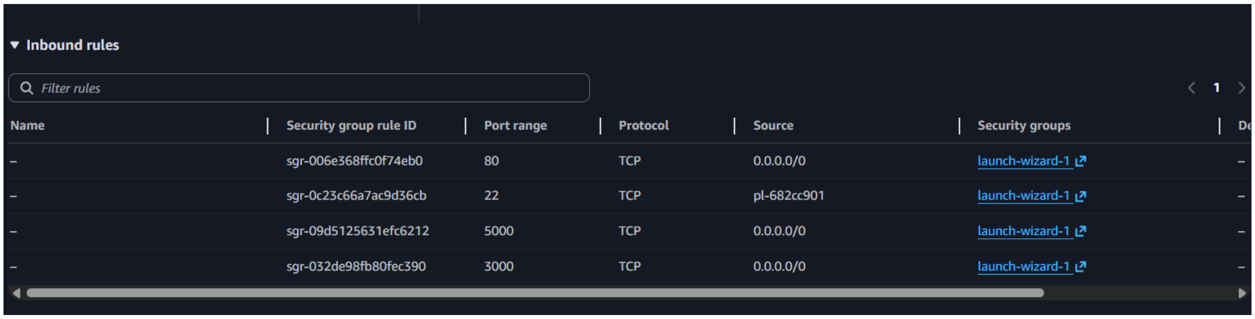
Public IP: **16.171.137.70**

3.2 Security Group Configuration

Ports allowed:

Port	Purpose
22	SSH
80	HTTP
3000	Express Frontend
5000	Flask Backend

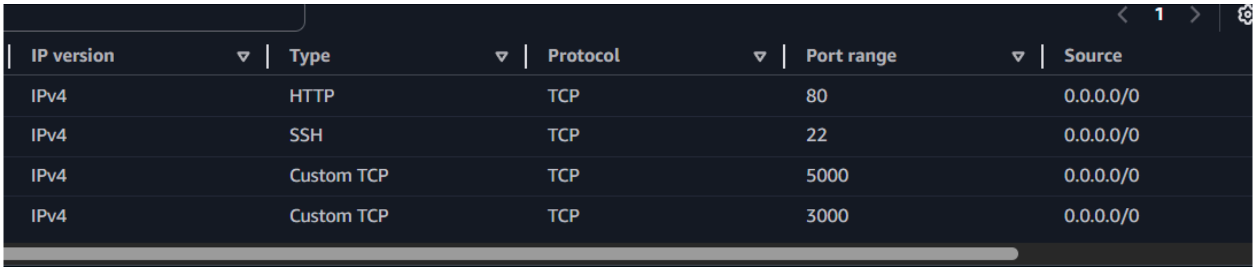
Security Group Rules



▼ Inbound rules

Name	Security group rule ID	Port range	Protocol	Source	Security groups	De
–	sgr-006e368ffc0f74eb0	80	TCP	0.0.0.0/0	launch-wizard-1	–
–	sgr-0c23c66a7ac9d36cb	22	TCP	pl-682cc901	launch-wizard-1	–
–	sgr-09d5125631efc6212	5000	TCP	0.0.0.0/0	launch-wizard-1	–
–	sgr-032de98fb80fec390	3000	TCP	0.0.0.0/0	launch-wizard-1	–

3.3 EC2 Launch Confirmation



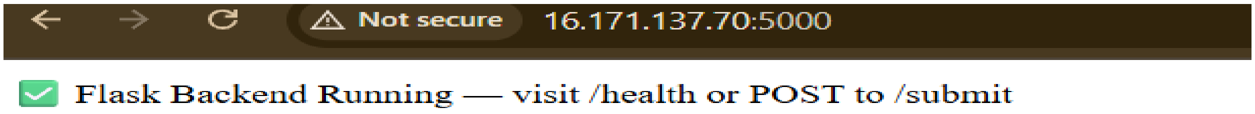
IP version	Type	Protocol	Port range	Source
IPv4	HTTP	TCP	80	0.0.0.0/0
IPv4	SSH	TCP	22	0.0.0.0/0
IPv4	Custom TCP	TCP	5000	0.0.0.0/0
IPv4	Custom TCP	TCP	3000	0.0.0.0/0

3.4 Flask Backend Deployment

Command used:

```
python3 app.py
```

Backend accessible at:
👉 **http://16.171.137.70:5000**



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

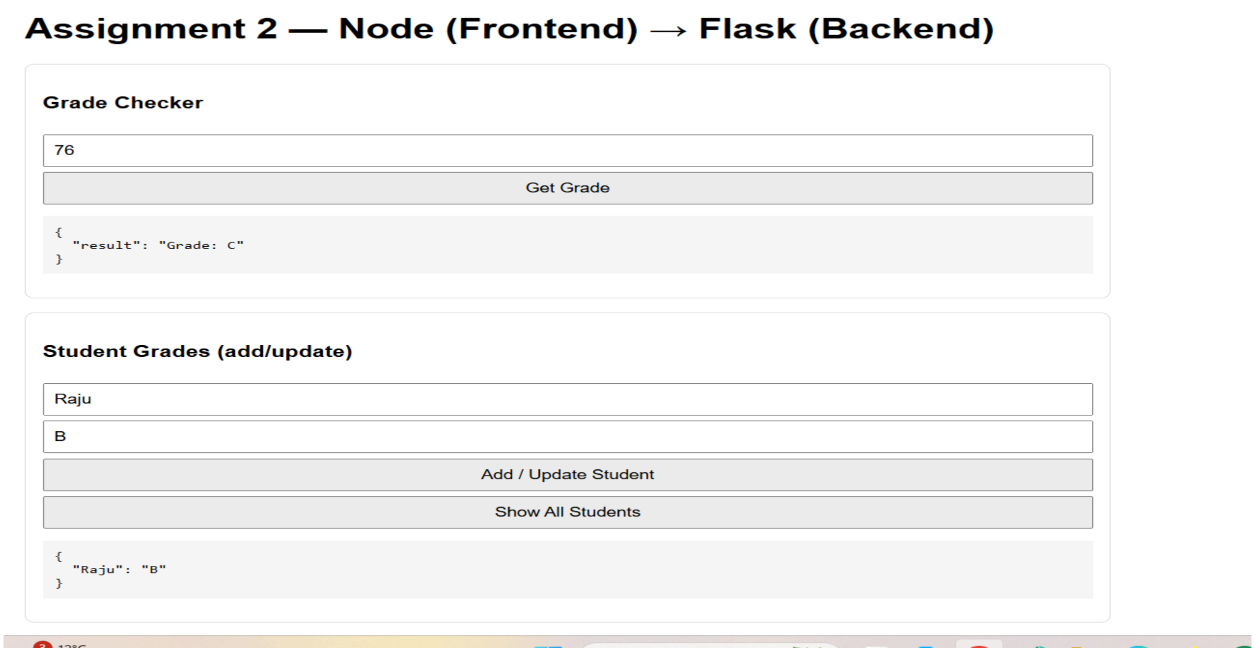
3.5 Express Frontend Deployment

Command used:

```
node server.js
```

Frontend accessible at:

👉 **http://16.171.137.70:3000**



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

Grade Checker

76

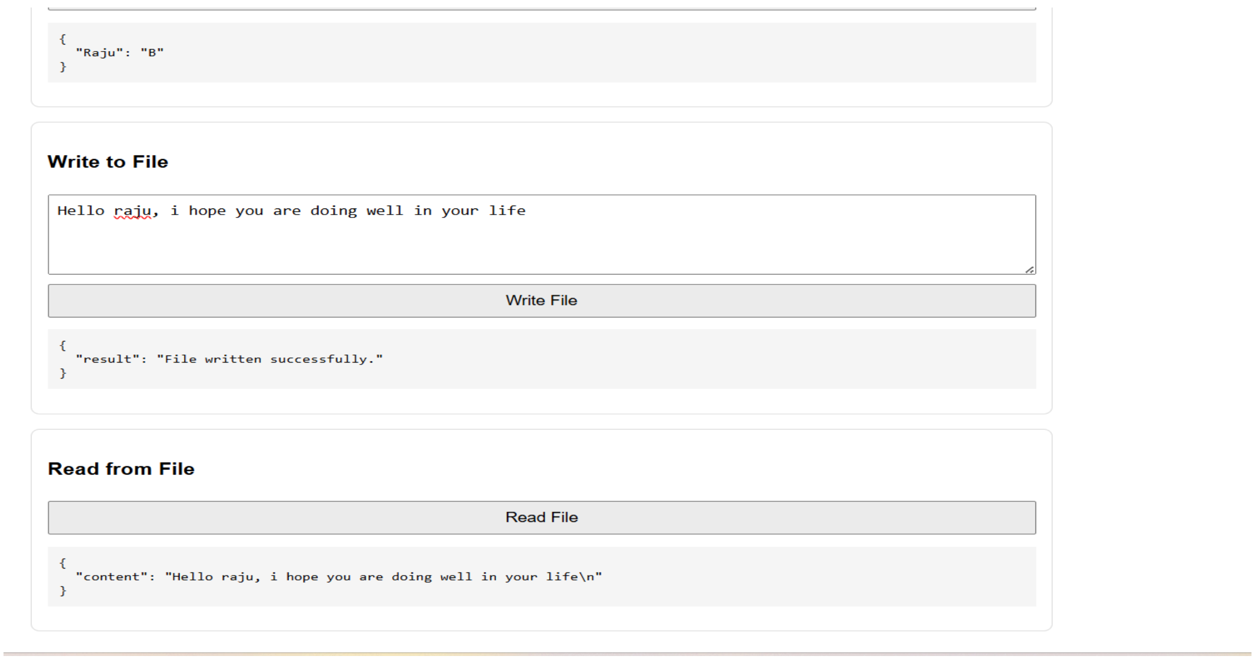
```
{
  "result": "Grade: C"
}
```

Student Grades (add/update)

Raju

B

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



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

Write to File

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

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

Read from File

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

4. Deployment on Separate EC2 Instances

In this architecture, the backend and frontend run on **two different EC2 instances**.

4.1 Backend EC2 Instance

- Flask installed
- Backend running on port **5000**
- SG allows port 5000

4.2 Frontend EC2 Instance

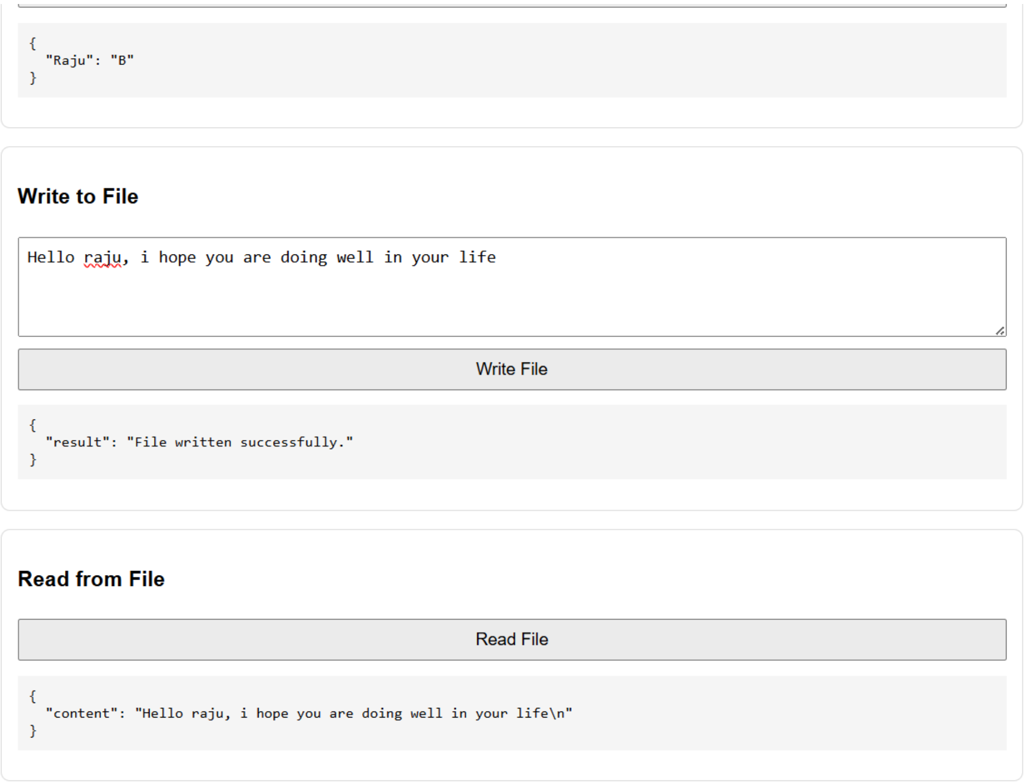
- Node.js installed
- API URL updated to backend EC2 IP
- SG allows port 3000

4.3 Communication

Frontend → Backend using:

```
http://<BACKEND_PUBLIC_IP>:5000
```

4.4 Verification Screenshot



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

Write to File

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

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

Read from File

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

---

5. Dockerized Deployment Using AWS ECR, ECS & VPC

This modern deployment uses **Docker containers**, **AWS ECR**, **ECS Fargate**, and **VPC networking**.

5.1 Docker Containerization

Images built:

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

Images tagged with ECR URIs.

5.2 Push Images to AWS ECR

1. Created private repositories in ECR
2. Logged in to ECR
3. Pushed Docker images

Your earlier PDF already contains these screenshots.

5.3 ECS Cluster Setup

- ECS cluster created
- Fargate launch type
- Public subnets & internet gateway assigned

5.4 Task Definitions & Services

- Two tasks: backend + frontend
- Ports 5000 and 3000 exposed
- Public IP enabled
- Correct IAM role used

5.5 Launching Containers

ECS pulled images from ECR and launched them.
Containers entered **RUNNING** state.

5.6 Application Verification

Both backend and frontend were reachable via ECS-assigned public IP.

6. Issues Faced & Resolutions

Issue	Resolution
Wrong ECR region used	Re-tagged images and pushed again
ECS could not pull images	Fixed ECR URI
LinkedRole missing	Recreated automatically
Frontend couldn't reach backend	Opened backend SG properly

7. Final Output Summary

Component	Status
Flask Backend	✓ Deployed
Express Frontend	✓ Deployed
Single EC2 Deployment	✓ Successful
Dual EC2 Deployment	✓ Successful
Docker + ECR + ECS Deployment	✓ Successful
AWS Networking (VPC)	✓ Configured

8. Conclusion

This assignment successfully demonstrates three different cloud deployment strategies on AWS.
The project provided practical experience in:

- EC2 provisioning

- Networking & security groups
- Docker containerization
- ECR (private image registry)
- ECS Fargate orchestration
- Debugging and monitoring cloud deployment

This hands-on exercise closely aligns with modern DevOps and cloud-native application deployment practices.