

Project Title:

Containerized Node.js Application Deployment on AWS ECS using Docker, ECR and ECS.

1. Introduction

This project demonstrates the complete procedure of deploying a Node.js application using Docker containers on AWS ECS. The workflow includes:

- Local development and testing
- Docker containerization
- Push to AWS ECR
- ECS cluster, task, and service configuration
- Health check and verification

The goal is to deploy a fully functional containerized application accessible via a public IP.

2. Project Overview

- **Application:** Node.js with Express framework
- **Container Port:** 8000
- **AWS Services Used:**
 - ECR (Elastic Container Registry)
 - ECS (Elastic Container Service) with Fargate
 - Security Group (port 8000 open)

3. Procedure

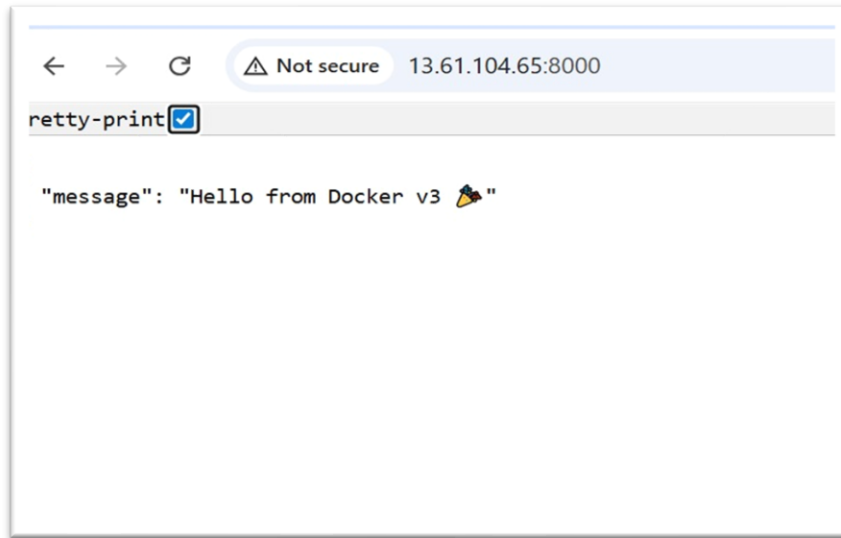
Step 1: Code Setup and Local Testing

- git clone [github repo](#) Ali
- cd Ali

Step 2: Docker Image Creation

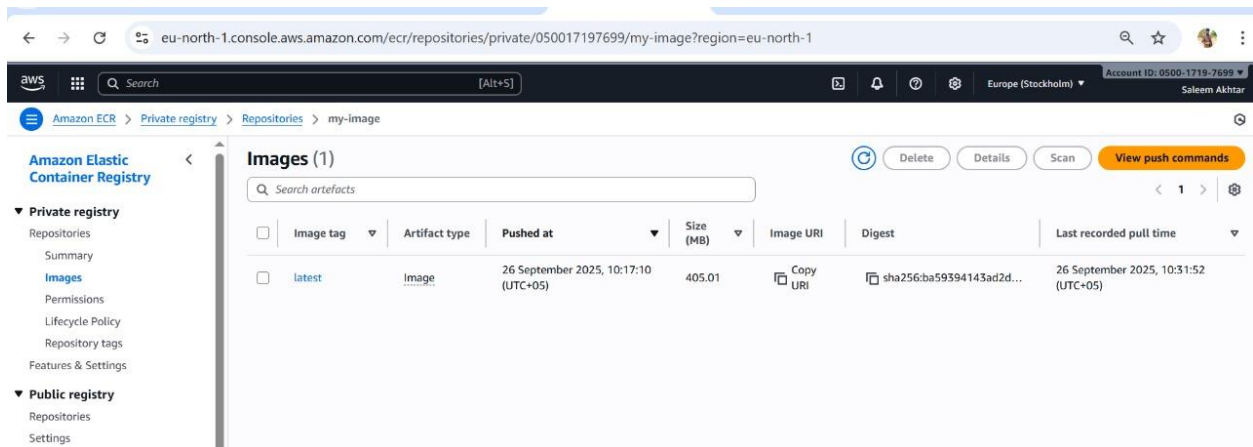
- docker build -t my-image .
- docker run -p 8000:8000 my-image

Step 3: Verify the app via browser



Step 3: Push Docker Image to AWS ECR

- Create an **ECR repository** in AWS Console.
- Go to repo name and view push commands on terminal



Step 4: ECS Cluster Creation

- Create a new cluster in ECS (Fargate, serverless).
- Select **VPC**, **subnet**, and security group allowing **port 8000**.

my-cluster-task:1

Overview

ARN: arn:aws:ecs:eu-north-1:05001719769:task-definition/my-cluster-task:1

Status: **ACTIVE**

Time created: 25 September 2025, 18:56 (UTC+5:00)

App environment: Fargate

Task role: -

Task execution role: [ecsTaskExecutionRole](#)

Operating system/Architecture: Linux/X86_64

Network mode: awsvpc

Fault injection: ☐ Turned off

Containers | JSON | Task placement | Volumes (0) | Requires attributes | Tags

Task size

Task CPU: 1,024 units (1 vCPU)

Task memory: 3,072 MiB (3 GB)

Task CPU maximum allocation for containers

Task memory maximum allocation for container memory reservation

CPU (unit) bar chart: 0 to 1000

Memory (MiB) bar chart: 0 to 2800

Step 5: Task Definition

- Define a new task using the ECR image.
- Set **container port 8000**.
- Enable **Health Check**:

Step 6: Service Creation

- Create an ECS service using the task definition.
- Set desired tasks = 1, launch type = FARGATE, platform version = LATEST.
- Assign public IP for external access.
- Wait for task status = HEALTHY.

my-cluster-task-service

Service overview

Status: **Active**

Tasks (1 Desired): 0 pending | 1 running

Task definition: revision [my-cluster-task:3](#)

Deployment status: **Success**

Health and metrics | **Tasks** | Logs | Deployments | Events | Configuration and networking | Service auto scaling | Tags

Tasks (1/1)

Filter tasks by property or value

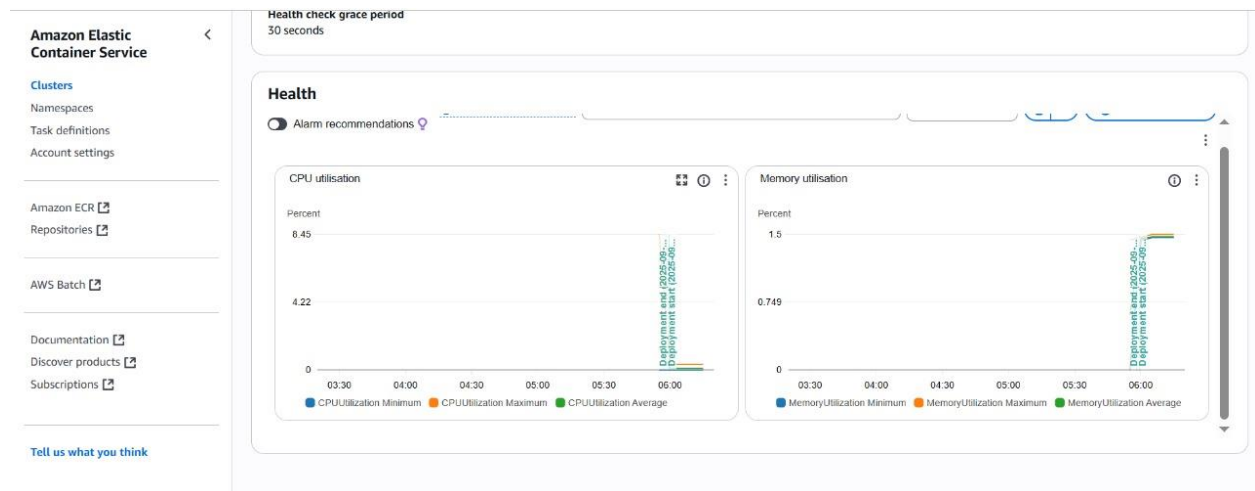
Filter desired status: Any desired status

Filter launch type: Any launch type

Task	Last status	Desired st...	T...	Health sta...	Created at	Started by	Started at
0806e5d4b6e0445d8681...	Running	<input checked="" type="checkbox"/>	Running	Healthy	3 minutes ago	ecs-svc/47940633752...	2 minutes ago

Containers for task 0806e5d4b6e0445d868116cffa22e619

ECS Cluster Health.



Step 7: Verification

- Access the app using **task public IP** on port 8000.
- Confirm ECS service **desired tasks** = **running**, and container health is **HEALTHY**.

```
{  
  "message": "Hello from Docker v3"  
}
```

5. Conclusion

This project demonstrates the complete end-to-end deployment of a Node.js application using Docker containers on AWS ECS. It highlights the process of containerizing an application, pushing Docker images to Amazon ECR, creating an ECS cluster along with task definitions and services, configuring health checks and security groups, and successfully deploying the application for external access. The workflow presented in this project serves as a reusable blueprint for deploying other containerized applications on AWS.