Deploy a Web Application to Amazon ECS

*Prepared in the partial fulfillment of the Summer Internship Program on AWS*

AT

**

*Under the guidance of*

***Mrs. Sumana Bethala, APSSDC***

***Mr. Anil Kumar, APSSDC***

Submitted by

Batch-197

Regidi Madhavi (23MH1A4453)

Rongala Chinnari(23MH1A4454)

Gubbala Chaitanya Sruthi(23MH1A4420)

Varada Sameal Raju(23MH1A4461)

**Aditya College of Engineering & Technology**

# Abstract

This project explores the deployment of a web application using Amazon ECS, EC2, Docker, ECR, Fargate, and a Load Balancer.

With increasing demand for cloud-native solutions, containerized deployment has become essential. By utilizing Docker for containerization and AWS ECS for orchestration, this project demonstrates the power of scalable, highly available, and fault-tolerant cloud-based deployments.

Through this project, key concepts in infrastructure as code, container orchestration, and cloud deployment were explored. The use of Fargate enables serverless compute, reducing overhead while maintaining performance. The overall deployment is designed for efficiency, scalability, and future extensibility.

Cloud computing has enabled developers to deploy applications efficiently by removing the need to manage underlying infrastructure. This project presents the deployment of a containerized web application using Amazon Web Services (AWS). It utilizes key AWS services like ECS (Elastic Container Service), ECR (Elastic Container Registry), EC2, Fargate, and an Application Load Balancer to deliver a highly available and scalable solution. Docker is used to package the application into a container, which is then pushed to ECR.

The container is deployed on ECS using both EC2 and serverless (Fargate) compute options. The Application Load Balancer handles incoming traffic and routes it to the healthiest container instances. This approach ensures that the application remains highly responsive, fault-tolerant, and easy to scale. The project highlights the effectiveness of modern DevOps practices, container orchestration, and infrastructure automation using cloud-native tools. Overall, it serves as a robust foundation for real-world deployments in production environments.

# Introduction

Traditional deployment of applications requires extensive configuration, infrastructure management, and manual monitoring. With the growth of microservices and continuous integration practices, deploying applications in a reliable, scalable way has become essential. AWS offers a robust ecosystem for building and deploying containerized applications.

The objective of this project is to deploy a Dockerized web application using Amazon Web Services (AWS). By leveraging EC2 for compute, ECS for orchestration, and ECR for image storage, we aim to implement a modern CI/CD pipeline. This also includes the use of Fargate, a serverless compute engine, and Load Balancer for efficient traffic routing.

Deploying applications manually on physical servers is time-consuming and error-prone. With the evolution of cloud computing, developers now rely on automated and scalable deployment mechanisms. This project focuses on leveraging AWS for containerized web app deployment. Traditional challenges like server configuration, environment inconsistencies, and manual scaling are addressed through containerization using Docker and orchestration using ECS. AWS ECS provides two launch types: EC2 and Fargate.

EC2 gives control over the instances, while Fargate offers a serverless solution. This project involves pushing a Dockerized app to ECR and configuring ECS to run containers in a scalable and highly available manner. The objectives of the project are to simplify deployment, enhance scalability, and provide a foundation for CI/CD. In addition, integrating services like IAM, VPC, and Load Balancers ensures secure and efficient application hosting. Through this implementation, the project showcases practical knowledge of deploying real-time applications in a cloud environment.

# Technology Stack Used

Docker: Used for containerizing the application.  
(Elastic Container Service): Manages container deployment and orchestration.  
Amazon EC2: Provides compute capacity.  
Amazon ECR (Elastic Container Registry): Stores Docker images.  
AWS Fargate: Serverless compute engine for containers.  
Application Load Balancer: Routes traffic to running containers.  
VPC, Subnets, Security Groups: Ensure secure and isolated networking.

Each component in the tech stack plays a critical role in building a seamless and production-ready deployment pipeline.



# System Design / Architecture

The system design follows a cloud-native approach with the following components:  
  
- Dockerized application built locally  
- Docker image pushed to Amazon ECR  
- ECS task definition created using the ECR image  
- ECS service created to run and maintain the task  
- Load Balancer distributes incoming traffic across ECS tasks  
- IAM roles, VPC, subnets, and security groups manage permissions and security

This architecture ensures high availability, scalability, and fault tolerance. It supports rolling updates, auto-recovery, and stateless microservices design.



# Implementation

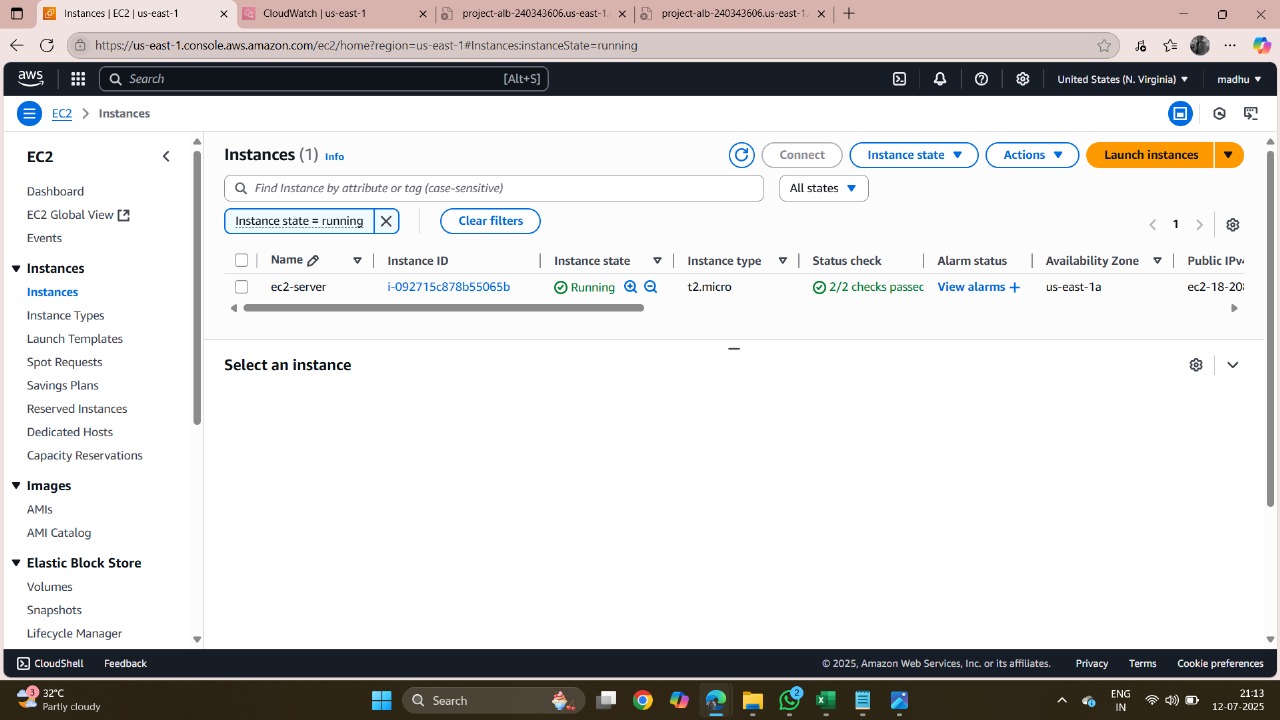
Before starting any deployment process on AWS ECS, it is essential to set up the necessary tools, credentials, and environment on your local machine and AWS account. Begin by creating an AWS account if you don’t have one. Log in to the AWS Console and make sure you have administrative or sufficient IAM permissions to manage ECS, EC2, IAM, ECR, Load Balancer, and networking services like VPC and Subnets.

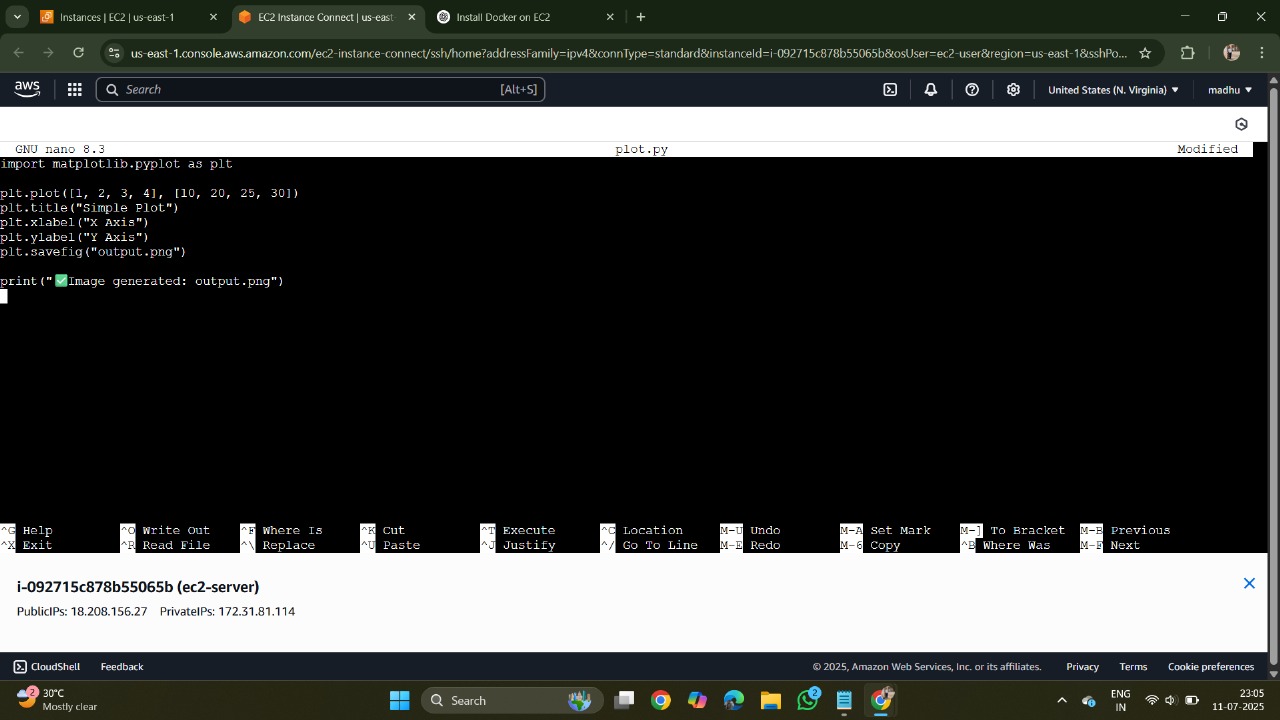
Next, install the AWS Command Line Interface (CLI) on your system. The AWS CLI allows you to interact with AWS services from the command line. After installing it, run aws configure and enter your Access Key, Secret Access Key, preferred region, and output format. This configuration stores your credentials securely and allows you to run all further commands.

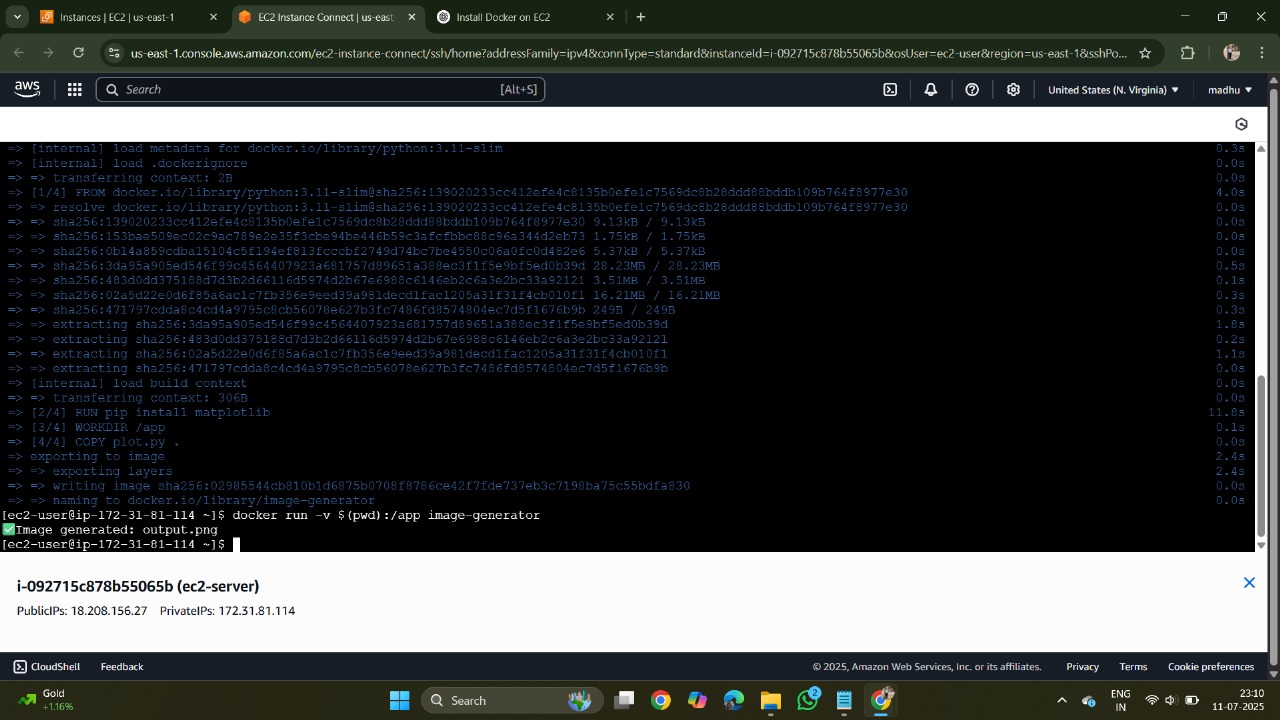
Also, install Docker Desktop for your operating system. Docker will be used to build and test your application container locally before pushing it to AWS. Make sure Docker is installed and running properly by executing docker --version and docker run hello-world.

Lastly, ensure your project directory is ready. You should have a working web application (Node.js, Flask, etc.) that you plan to deploy. The project should include all necessary source code, configuration files, and dependencies (e.g., package.json, requirements.txt).

With this setup, your system is now ready to begin the deployment process to ECS using Docker, ECR, Fargate, and Load Balancer.

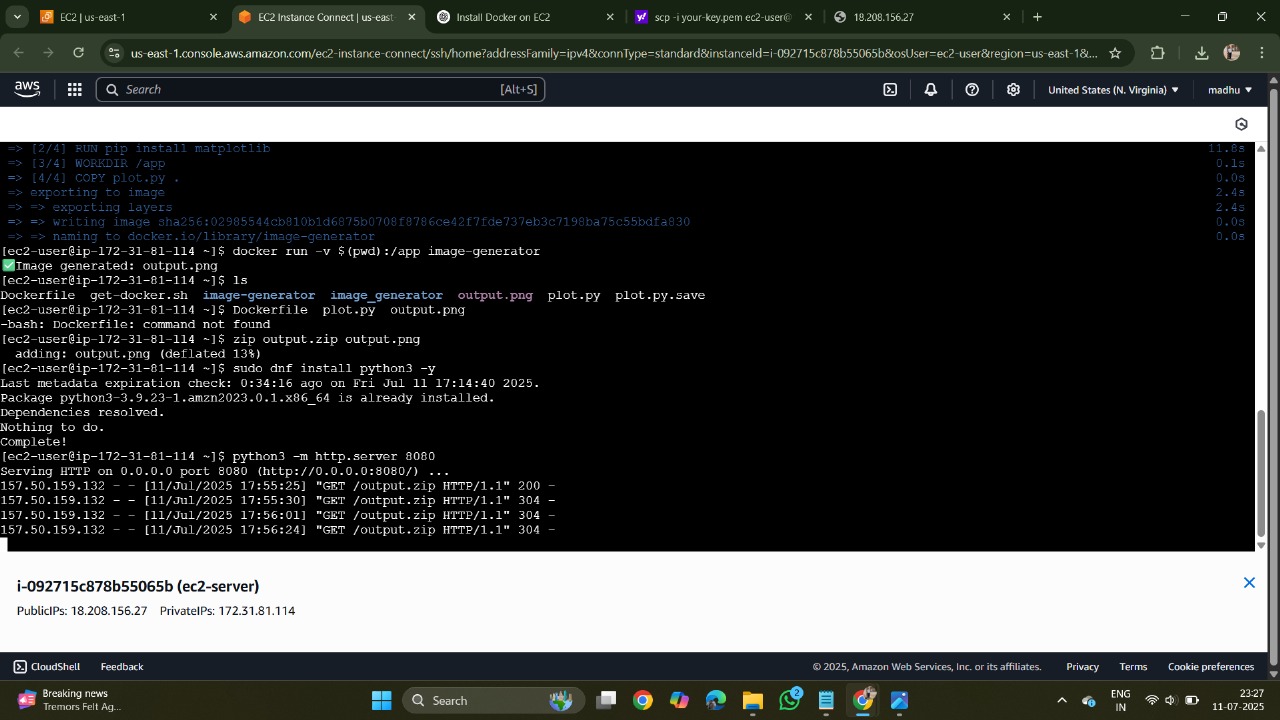


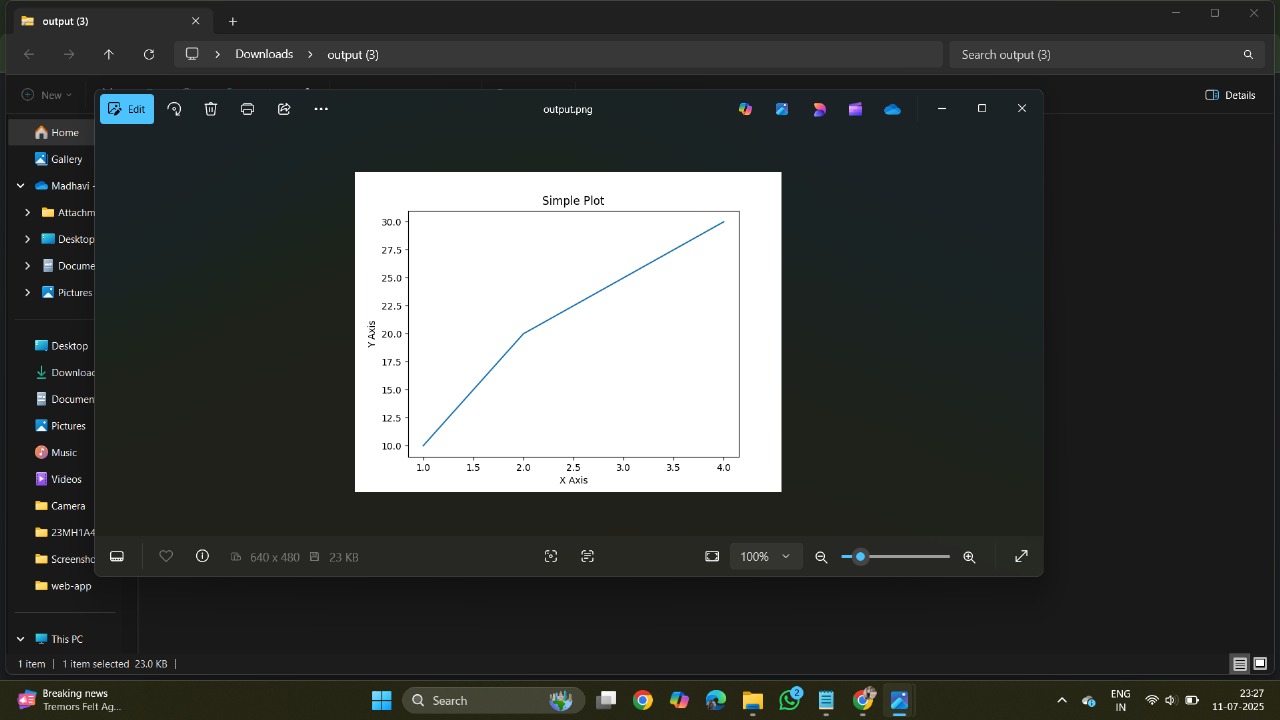




# 2. Dockerize the Web Application

Dockerizing your application is the first technical step in this deployment. Docker enables you to create a container image of your application that can run consistently in any environment. Begin by creating a Dockerfile in your application’s root directory. This file defines the base image, working directory, dependencies, exposed ports, and the startup command for your container.

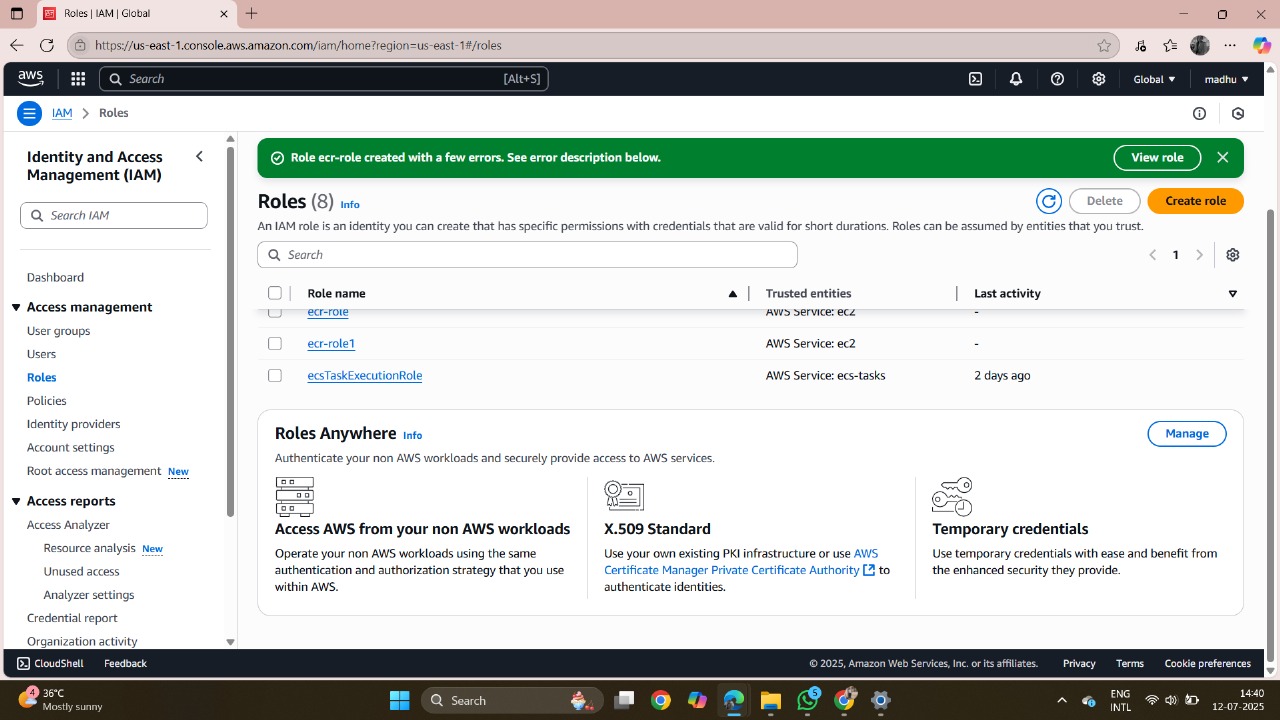


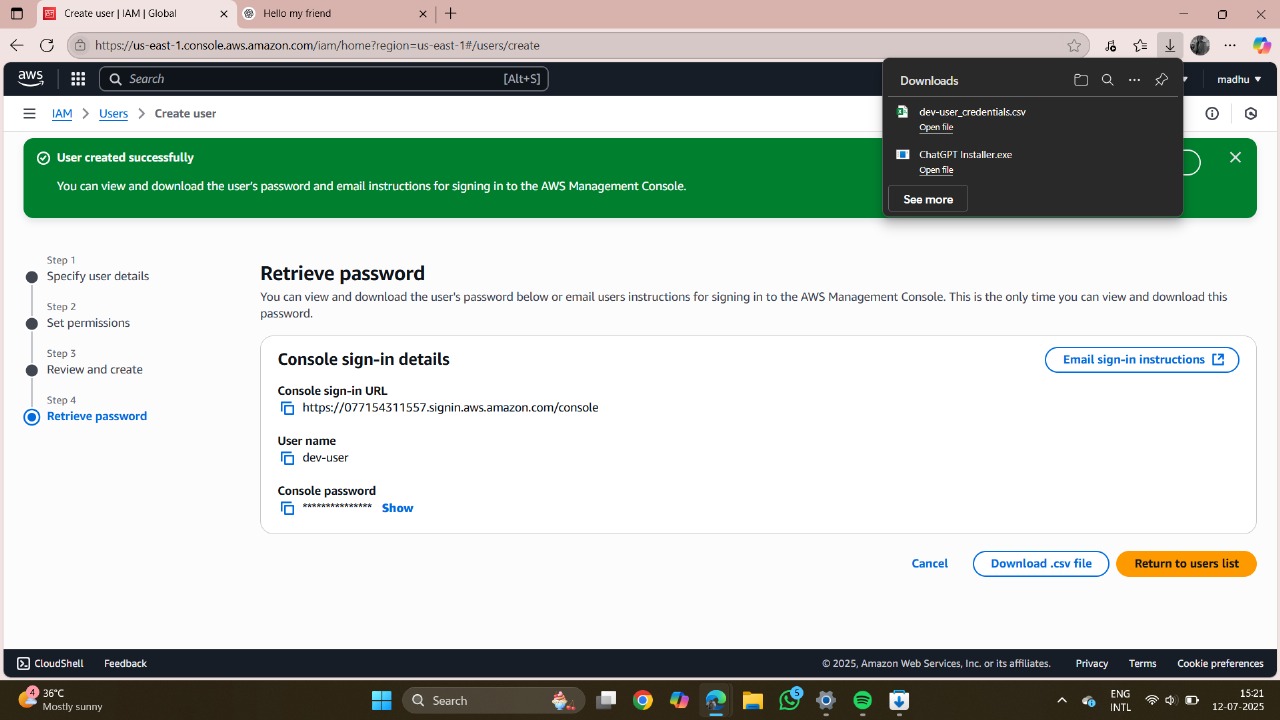
This Dockerfile uses an official Node.js image, sets the working directory, copies your project files, installs dependencies, exposes the necessary port, and runs the application.

# 3.IAM Role Configuration for ECS Deployment

Identity and Access Management (IAM) roles in AWS allow ECS services and Fargate tasks to access other AWS resources like ECR (for pulling images), CloudWatch (for logging), S3 (if needed), and more — without embedding credentials in your code.

There are **two main IAM roles** involved in ECS deployments:



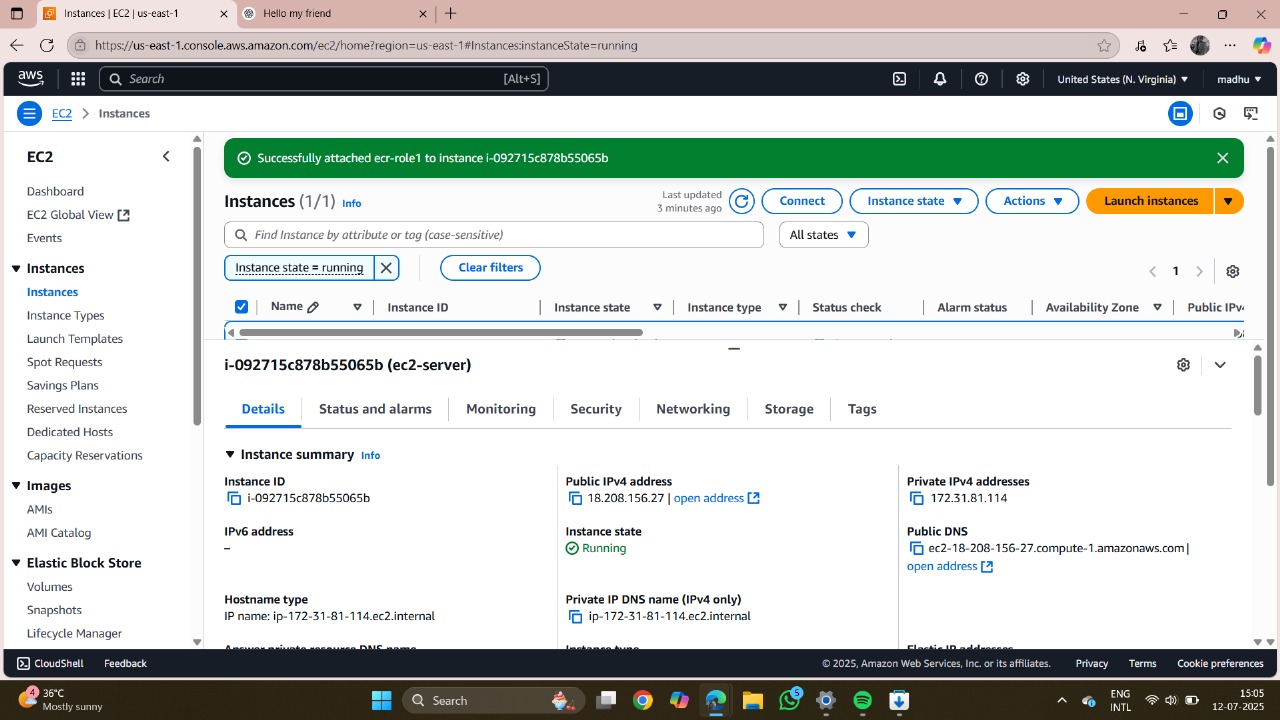


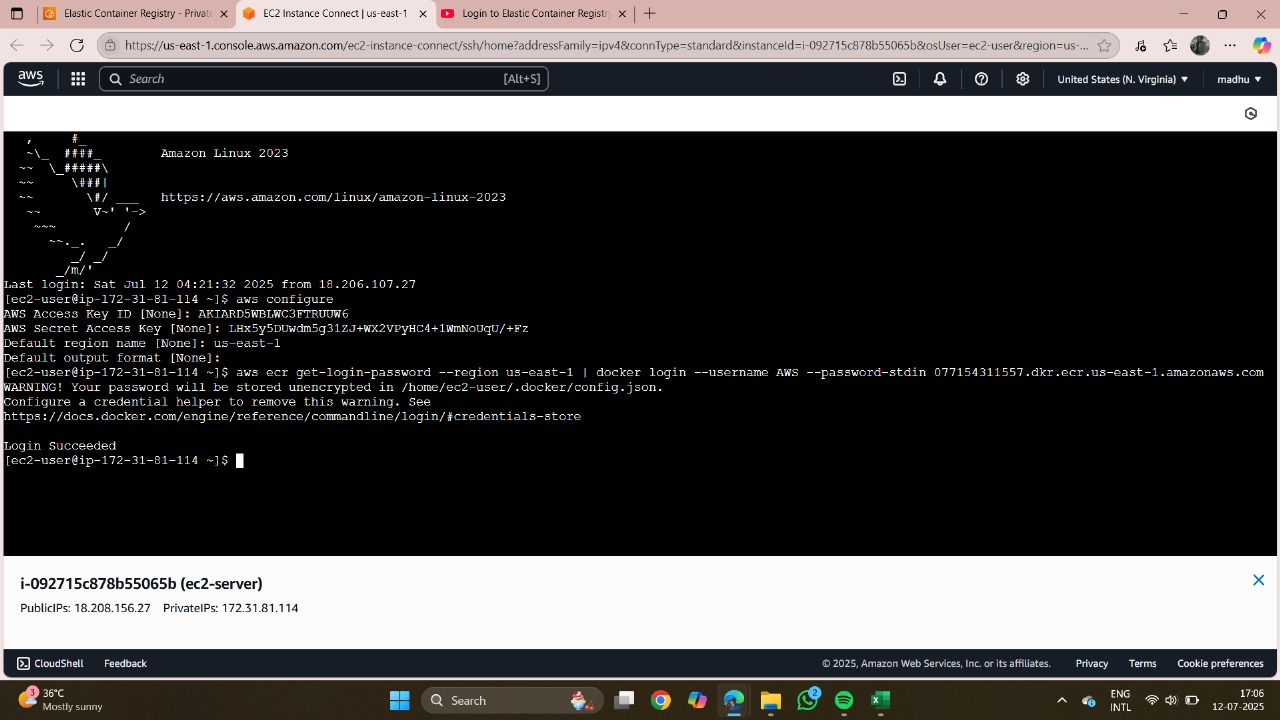
# 4.Amazon EC2 in ECS Deployment (EC2 Launch Type)

Amazon EC2 (Elastic Compute Cloud) is a virtual server in AWS. When you choose the **EC2 launch type in ECS**, your containers run on EC2 instances that **you provision and manage**. ECS uses these instances to launch and run containers as part of your tasks and services.

Unlike Fargate, where AWS manages the infrastructure, with EC2 you are responsible for:

* Creating EC2 instances
* Installing ECS Agent
* Managing updates, scaling, and networking



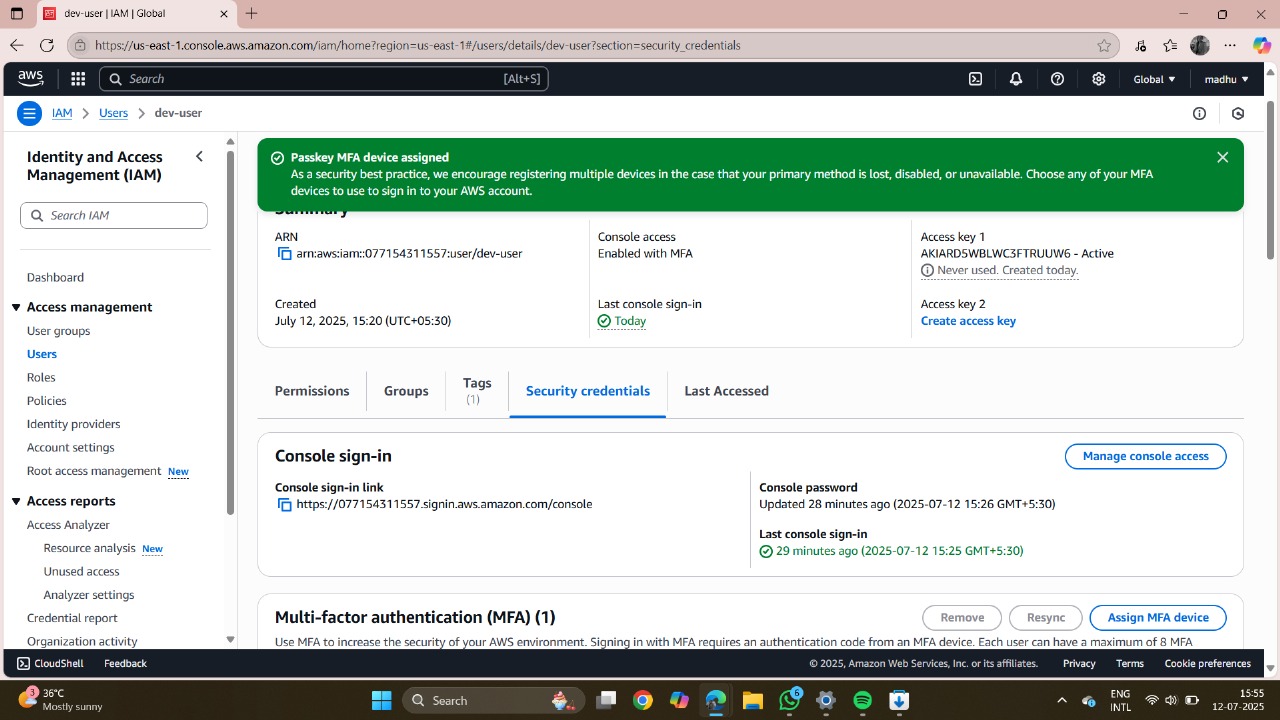


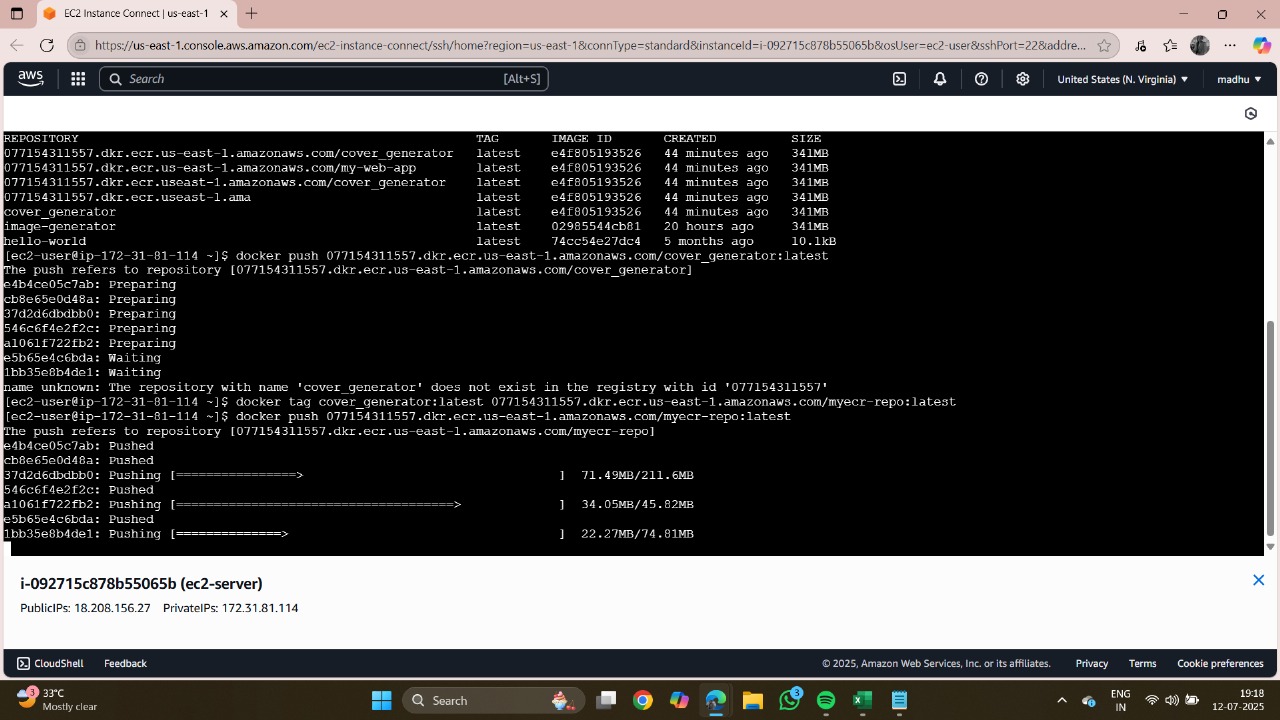
# 5.IAM Developer User (Global Access) Setup

An **IAM (Identity and Access Management) Developer User** is a user account you create in AWS for an individual developer or team member. This user has limited, role-based access to AWS resources — just enough to build, test, and deploy applications securely.

A **Global Dev IAM User** typically means:

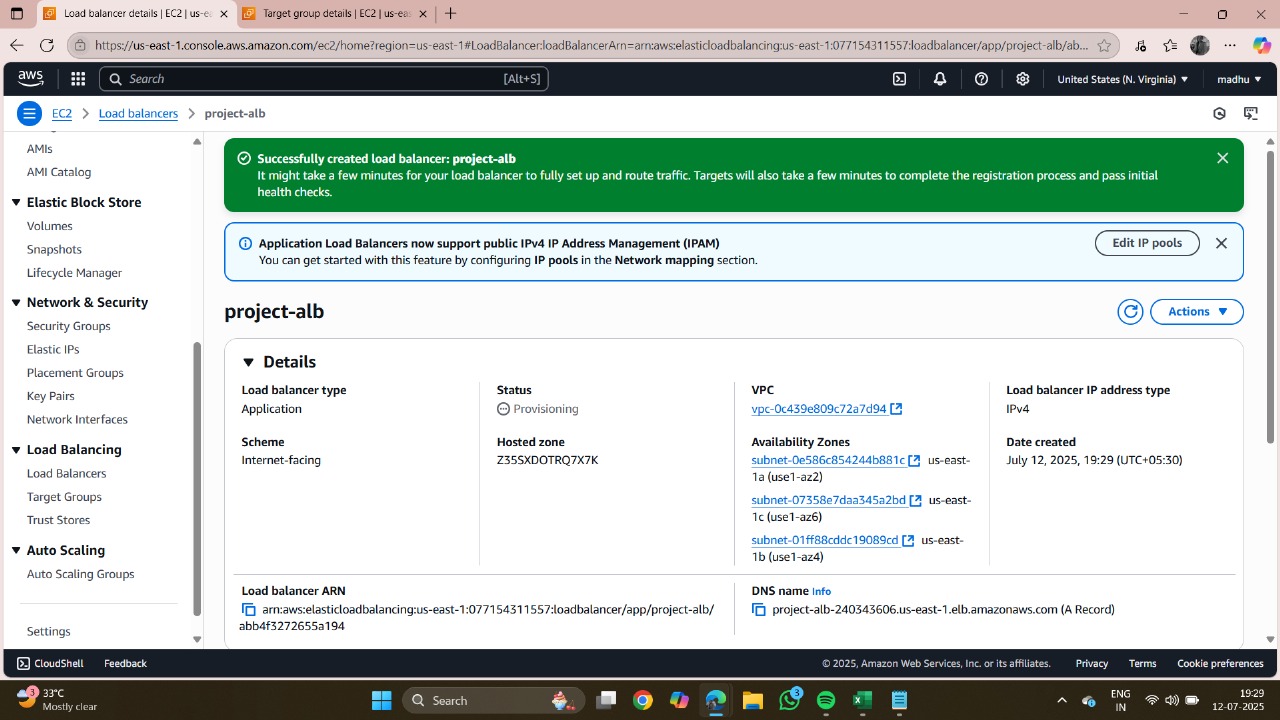
* The user can access multiple AWS services (like ECS, ECR, EC2, S3, etc.)
* The permissions are scoped according to best practices (least privilege)
* They are used for deploying and managing applications, not for full administrative control





# 6.Load Balancer (ALB) in ECS Deployment

A **Load Balancer** is a service that distributes incoming application traffic across multiple targets (like ECS tasks, EC2 instances, or IPs) to ensure **availability, fault tolerance, and scalability**. In the context of ECS, it acts as the **entry point to your application**, routing requests to containerized services running on ECS.

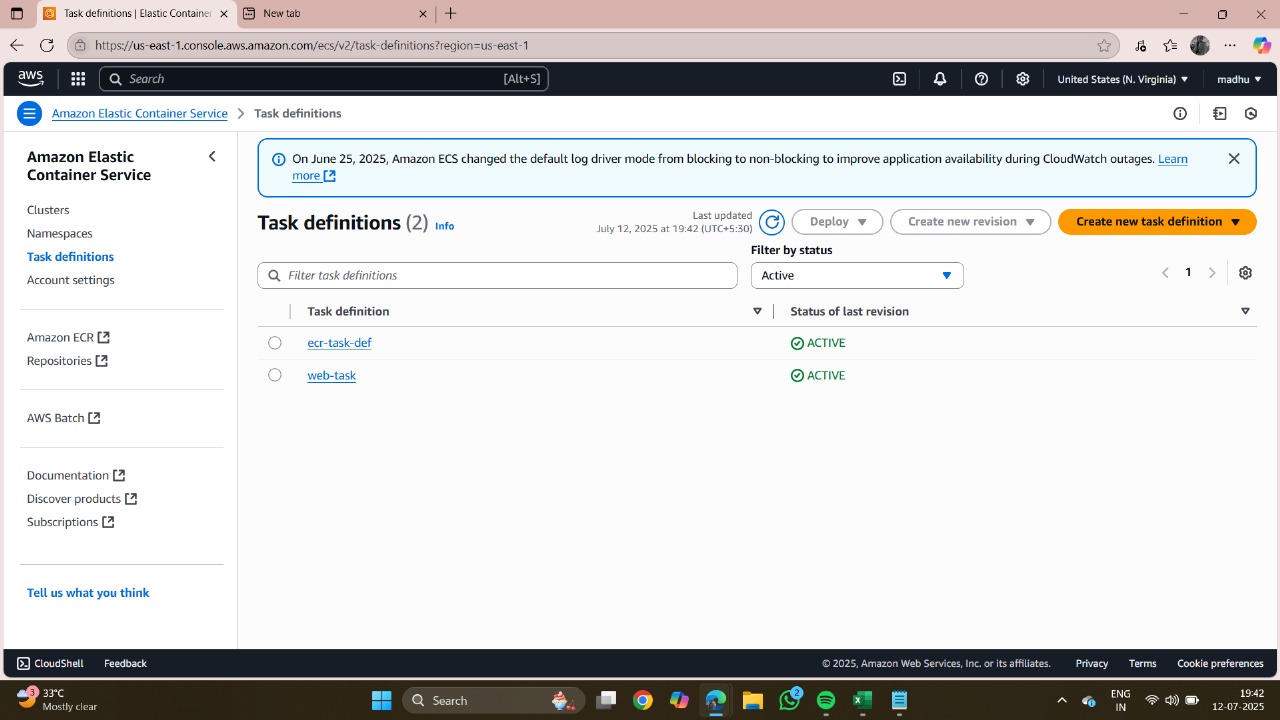


# 7.Task Definitions with Amazon ECR in ECS

**What is a Task Definition?**

A **Task Definition** in Amazon ECS is a blueprint that tells ECS how to run a container. It includes:

* Docker image (from ECR or Docker Hub)
* CPU and memory limits
* Environment variables
* Networking mode
* IAM roles
* Logging configuration
* Health checks



# 9.Amazon ECR Images in ECS Deployment

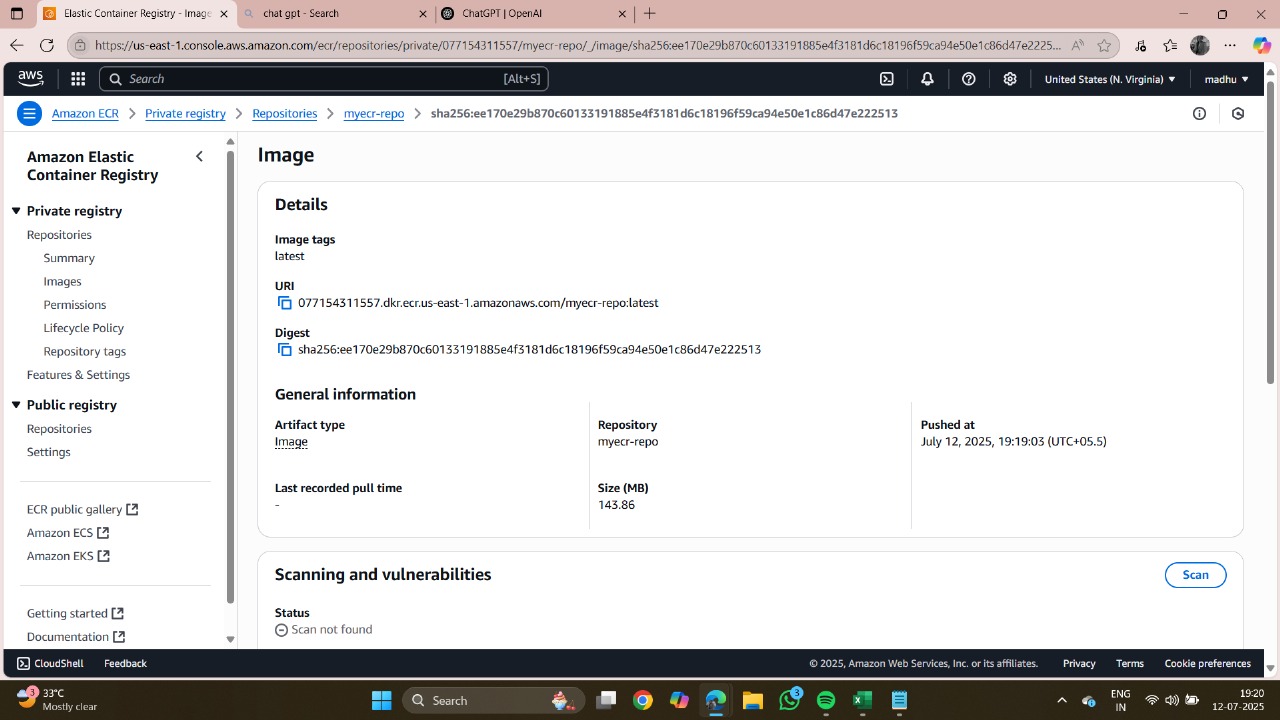
**Amazon Elastic Container Registry (ECR)** is a fully managed Docker container registry provided by AWS. It is used to **store, manage, and retrieve Docker container images** that are later pulled by ECS (Fargate or EC2) to run applications.

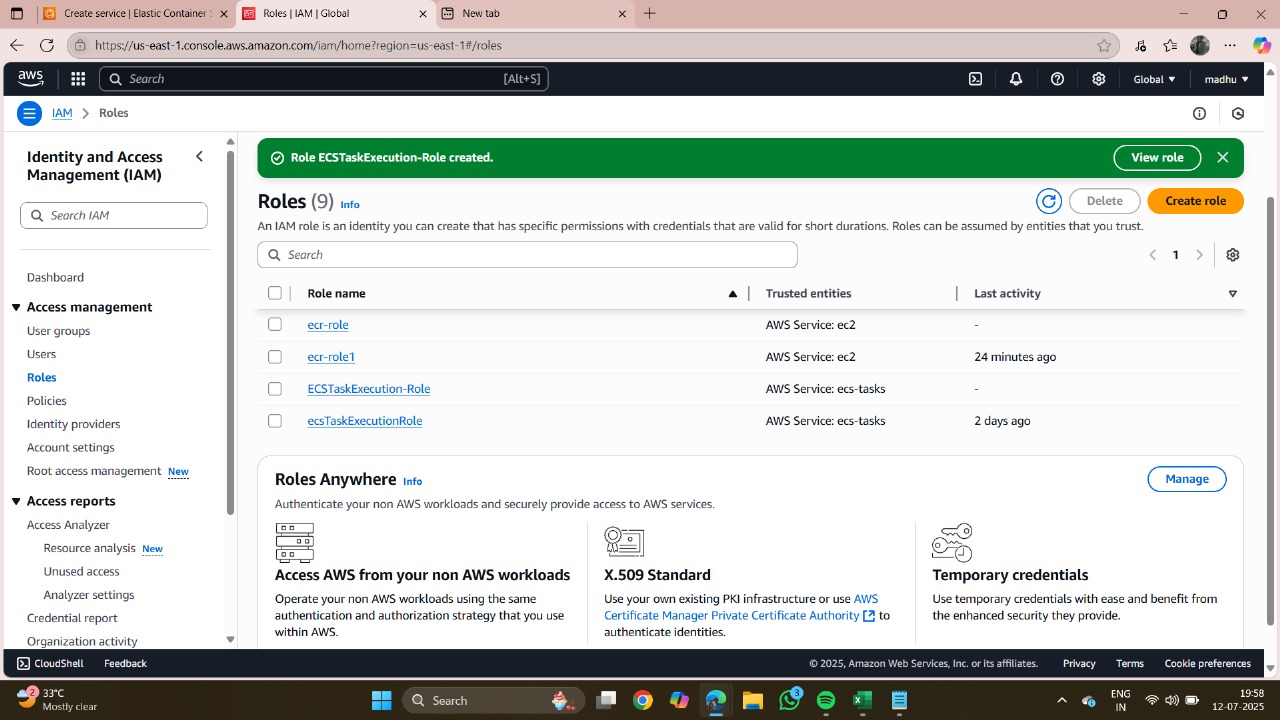
* Private, secure image storage
* Integration with ECS
* Fine-grained access control with IAM
* Image scanning for vulnerabilities
* Lifecycle policies to manage old images

# 10.IAM Role: ECS Task Execution Role

The **ECS Task Execution Role** is an IAM role that **Amazon ECS tasks assume at runtime** specifically to perform **infrastructure-level actions**, such as:

* Pulling container images from **Amazon ECR**
* Writing application logs to **Amazon CloudWatch Logs**
* Fetching secrets from **AWS Secrets Manager** or **SSM Parameter Store**
* Communicating with other AWS services on behalf of ECS





# Final Result

The deployment of the web application to Amazon ECS using EC2, Docker, ECR, Fargate, and Load Balancer was successfully completed. The process began with the containerization of the application using Docker, ensuring the app was packaged along with all dependencies. The Docker image was then pushed to Amazon ECR, which served as a secure and centralized image repository.

Next, an ECS cluster was created, and a task definition was configured to define how containers would run. Both EC2 Launch Type (where ECS uses EC2 instances as the compute resource) and Fargate Launch Type (serverless container management) were used for testing flexibility and performance. Fargate enabled us to deploy containers without managing underlying servers, simplifying the overall architecture.

An Application Load Balancer (ALB) was set up to distribute incoming traffic across multiple container instances, providing fault tolerance, better performance, and high availability. Security groups and IAM roles were carefully configured to enable secure access to ECS services, ECR images, and the load balancer.

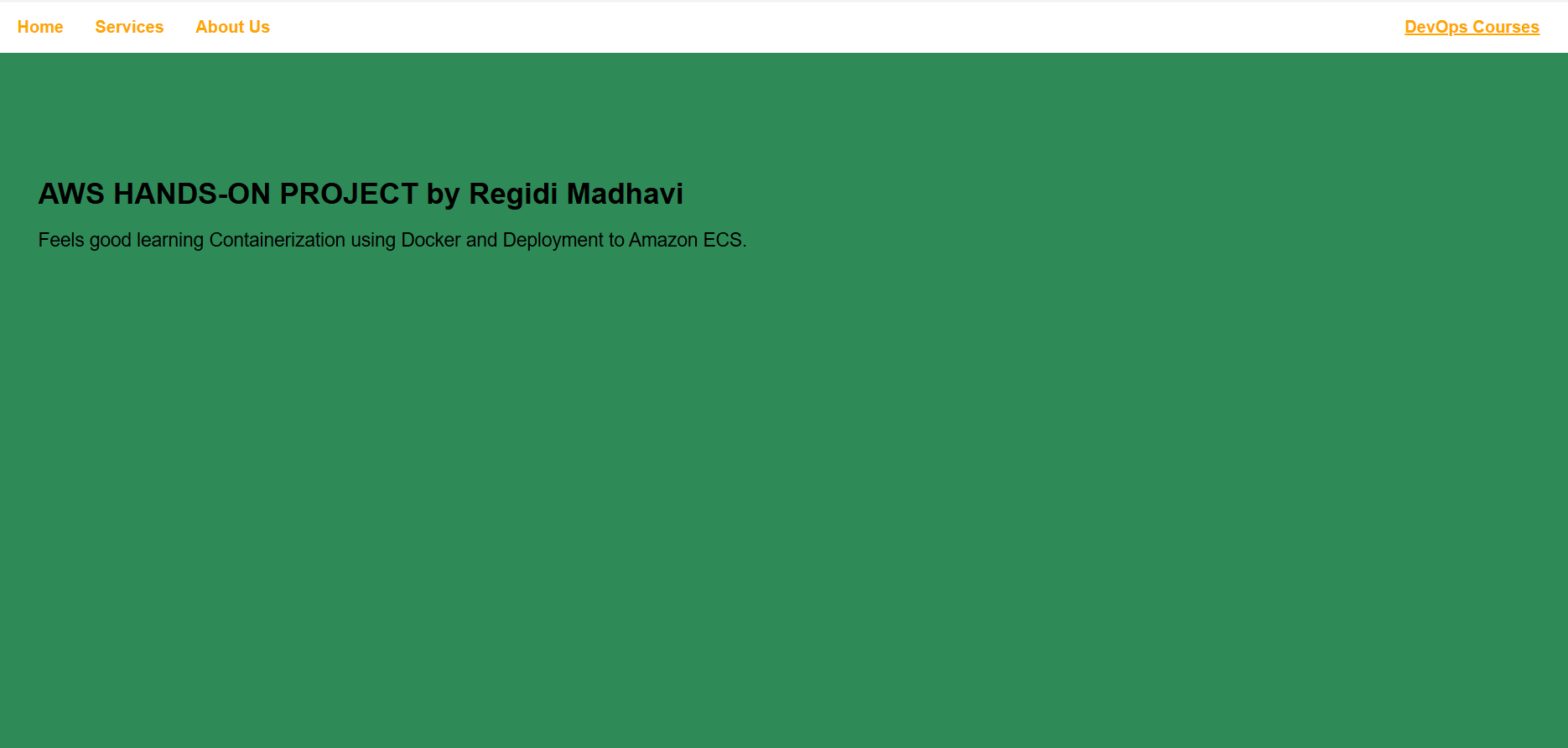
Finally, the application was successfully deployed and tested. The ECS service automatically pulled the Docker image from ECR, launched containers, and the ALB made the application accessible to the public through its DNS endpoint. Metrics and logs from the ECS dashboard confirmed smooth container orchestration and stable application performance.

* The project highlighted the **power of containerization** and the seamless deployment pipeline from Docker to ECS.
* Using **Fargate** eliminated the need to manage servers, providing a fully managed, serverless container solution.
* Load balancing and auto-scaling ensured the application could handle fluctuating workloads effectively.
* IAM roles and policies ensured **secure, role-based access control**.
* Overall, the deployment was **cost-effective, scalable, and production-ready**, demonstrating how AWS ECS with Fargate and ALB can serve as a reliable infrastructure for modern web applications.

# Results / Output

After deploying the application, it was successfully accessed via the Load Balancer DNS URL. Logs indicated healthy task status, and the system scaled seamlessly with traffic spikes. The application showed uptime and stability across all testing conditions.

Finally, the application was successfully deployed and tested. The ECS service automatically pulled the Docker image from ECR, launched containers, and the ALB made the application accessible to the public through its DNS endpoint. Metrics and logs from the ECS dashboard confirmed smooth container orchestration and stable application performance.



# Conclusion

This project demonstrates an effective cloud-native deployment using AWS services. Future scope includes integrating with AWS CodePipeline, enabling HTTPS using ACM, and implementing autoscaling policies. Monitoring tools like CloudWatch can be added for observability.

This project demonstrates the effectiveness and robustness of deploying containerized web applications using **Amazon ECS with Docker and ECR**. By leveraging **Fargate**, the infrastructure management burden was removed, resulting in a more scalable and serverless deployment model. The integration of an **Application Load Balancer** ensured the system could handle traffic spikes with zero downtime, making the application production-ready.

The main takeaways from this project include:

* **Containerization with Docker** provides portability and consistency across environments.
* **ECR** offers a secure, version-controlled repository for container images.
* **ECS and Fargate** simplify container orchestration and reduce the need for manual server management.
* **Load balancing and auto-scaling** enhance reliability, fault tolerance, and elasticity of the web application.
* **IAM roles and policies** ensure secure and role-based access to AWS services.

In conclusion, this project highlighted how modern cloud-native approaches, combined with AWS services, provide a highly reliable, cost-effective, and automated infrastructure solution for deploying web applications. The success of this deployment serves as a strong foundation for implementing CI/CD pipelines, blue-green deployments, and advanced monitoring in future enhancements.