Information Technology University Punjab

AWS Deployment and Microservices Implementation Report

A Technical Report on Continuous Deployment using Docker, ECS, and CodePipeline

Prepared by:

Hazira Azam Bsse23019

Areeba Shahbaaz Bsse23097

Supervisor:

Sir Zunnurain Hussain

Teaching Assistant: Sir Umair Makdoom

Contents

Abstract 5
1. Introduction 5
2. Problem Statement 5
3. Aim and Objectives 6
4. System Architecture and Design 7
4.1 Microservices Architecture 7
4.2 AWS Infrastructure Overview 8
4.3 Container Orchestration with Amazon ECS 11
4.4 Database Architecture 11
5. Implementation Details 13
5.1 Service Development 13
5.2 Containerization Strategy 15
6. CI/CD Pipeline Design and Automation 17
6.1 Pipeline Workflow 18
6.2 Blue-Green Deployment Strategy 19
7. Security and Networking 19
7.1 Identity and Access Management 19
8. Network Security 23
9. Limitations and Future Work 23
10. Conclusion 23

Tables of Figures:

Figure 1: Architecture Diagram 7
Figure 2: AWS Microservices Architecture of the Study Group & Peer Tutoring Platform 9
Figure 3: Configuration of the Virtual Private Cloud (VPC) for Network Isolation. 9
Figure 4: Enabling DNS Hostnames within the VPC Attributes. 10
Figure 5 Confirmation of DNS Resolution Support for Internal Service Discovery. 10
Figure 6: Subnet Architecture: Allocation of Public and Private Subnets across Availability Zones..... 11
Figure 7: Initialization of Database Schema and Table Structures in Amazon RDS 12
Figure 8: Database Connectivity and SQL Query Execution Verification. 12
Figure 9 :Amazon ECS Cluster Initialization and Infrastructure Selection (Fargate). 13
Figure 10: Task created 14
Figure 11 : Definition of Task Parameters (vCPU and Memory Allocation). 14
Figure 12: Configuration of Container Definitions and Port Mappings. 15
Figure 13: Deployment of ECS Service and Initial Task Running Status. 15
Figure 14: Amazon Elastic Container Registry (ECR) Private Repository Setup. 16
Figure 15: Execution of Docker Build Command via AWS CodeBuild. 16
Figure 16: Container Image Tagging and Versioning for Registry Submission. 17
Figure 17 Successful Push of Frontend/Backend Images to ECR. 17
Figure 18: Target Groups 18
Figure 19: IAM Role Customization and Association for ECS Task Execution. 20
Figure 20: Security Groups 20
Figure 21: Application Load Balancer (ALB) Configuration and Listener Rules. 21
Figure 22: Stress Testing: Simulating Concurrent User Traffic to the ALB. 21
Figure 23: Load Balancer Latency and Request Count Analysis. 22
Figure 24: Successful Response Validation under Peak Load Conditions. 22
Figure 25 : Final System Health Check and Blue-Green Deployment Completion with ALB 22

List of Tables:

Table 1: **: Core Microservices and Their Responsibilities** 8
Table 2: AWS Services Utilized in the Platform 12
Table 3: CI/CD Pipeline Stages and Responsibilities 17
Table 4: **Implemented Security Measures** 19

Abstract

This report presents the design, implementation, and deployment of a cloud-native Study Group & Peer Tutoring Platform developed using a microservices architecture on Amazon Web Services (AWS). The system addresses scalability, availability, and deployment challenges inherent in traditional monolithic academic collaboration platforms. All services were implemented in Node.js, containerized using Docker, and orchestrated through Amazon Elastic Container Service (ECS). A fully automated CI/CD pipeline was established using GitHub, AWS CodePipeline, CodeBuild, Amazon Elastic Container Registry (ECR), and AWS CodeDeploy, enabling Blue-Green deployments with zero downtime. The platform was deployed within a secure Virtual Private Cloud (VPC) architecture, utilizing public and private subnets, an Application Load Balancer (ALB), and an Amazon RDS database hosted in a private subnet. The results demonstrate improved scalability, operational reliability, and deployment efficiency, validating the suitability of microservices and DevOps practices for modern cloud-based educational platforms.

1. Introduction

The increasing reliance on digital platforms in higher education has significantly transformed the way students collaborate, access academic support, and engage in peer-assisted learning. While informal study groups and peer tutoring have long been recognized as effective learning strategies, existing digital solutions often fail to provide scalable, reliable, and well-coordinated environments. Many platforms suffer from rigid architectures, limited automation, and frequent service disruptions during maintenance or updates.

This project introduces a Study Group & Peer Tutoring Platform designed and deployed as a fully cloud-native system on AWS. The platform leverages microservices architecture to decompose core functionalities into independently deployable services, thereby improving scalability, fault isolation, and development agility. To support continuous delivery and operational excellence, a complete CI/CD pipeline was implemented, enabling automated builds, container image management, and zero-downtime deployments.

The system was developed as part of a Software Development & Construction initiative, emphasizing industry-aligned practices such as containerization, infrastructure security, automated testing, and DevOps-driven deployment workflows. By integrating AWS-managed services, the platform demonstrates how academic systems can achieve enterprise-grade reliability and performance while remaining cost-effective and maintainable.

2. Problem Statement

University students frequently encounter structural and technical challenges when attempting to organize collaborative learning activities. These challenges can be categorized into functional limitations and architectural constraints.

From a functional perspective, students often experience difficulty in discovering suitable tutors, coordinating study group sessions, and resolving scheduling conflicts across multiple participants. The absence of centralized and reliable platforms leads to fragmented communication, inefficient coordination, and reduced academic engagement.

From a technical standpoint, many existing systems rely on monolithic architectures that tightly couple application components. Such architectures introduce several critical issues:

- Limited scalability during peak academic periods, such as examinations
- High risk of system-wide failure due to single points of failure
- Inflexible deployment processes that require complete system downtime
- Increased maintenance complexity and slower feature delivery

Additionally, traditional deployment approaches lack automation, making updates error-prone and time-consuming. Service downtime during deployments directly impacts user trust and system usability, particularly in academic environments where availability is crucial.

3. Aim and Objectives

The primary aim of this project was to design, implement, and deploy a secure, scalable, and highly available Study Group & Peer Tutoring Platform using modern cloud-native technologies and DevOps practices.

To achieve this aim, the following objectives were defined:

1. **Microservices-Based Design**: Decompose the application into independent Node.js microservices to enhance modularity, fault tolerance, and scalability.
2. **Containerization**: Package all microservices using Docker to ensure environment consistency across development, testing, and production.
3. **Cloud Deployment on AWS**: Deploy the system on AWS using ECS for container orchestration, an Application Load Balancer for traffic distribution, and Amazon RDS for persistent data storage in a private subnet.
4. **CI/CD Automation**: Implement a complete CI/CD pipeline integrating GitHub, AWS CodePipeline, CodeBuild, ECR, and CodeDeploy to automate the build, test, and deployment lifecycle.
5. **Zero-Downtime Deployment**: Utilize a Blue-Green deployment strategy to ensure uninterrupted service availability during application updates.
6. **Security and Networking**: Enforce secure communication and access control using IAM roles, security groups, VPC isolation, and secure secret management.
7. **Operational Reliability**: Enhance observability and reliability through centralized logging, monitoring, and controlled scaling mechanisms.

These objectives collectively ensure that the platform aligns with industry best practices and demonstrates a practical application of cloud-native software engineering principles.

4. System Architecture and Design

The Study Group & Peer Tutoring Platform was designed using a cloud-native microservices architecture to ensure scalability, resilience, and independent service evolution. Rather than adopting a monolithic design, the system decomposes core functionalities into loosely coupled services, each responsible for a specific business domain. This architectural choice enables horizontal scaling, fault isolation, and continuous deployment without impacting the entire system.

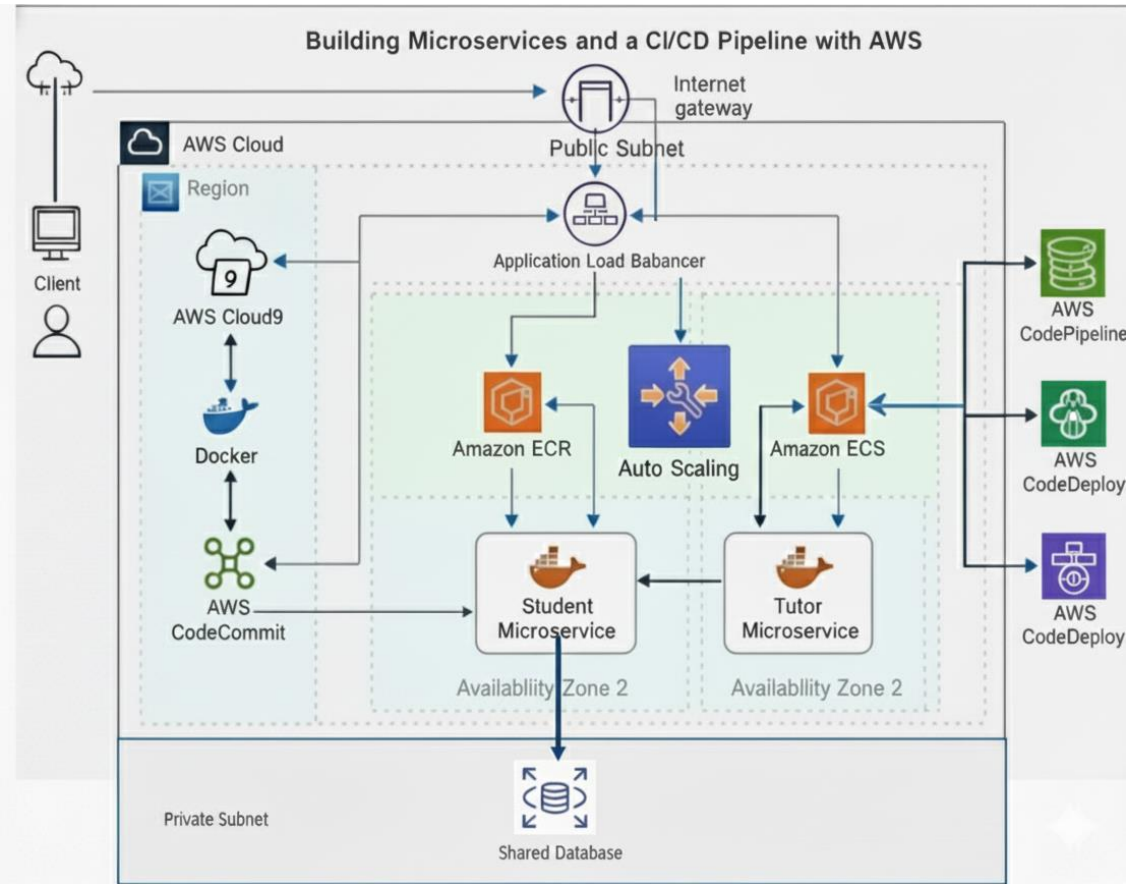

Figure 1: Architecture Diagram

4.1 Microservices Architecture

Each microservice was implemented using Node.js and exposes RESTful APIs to support inter-service communication. Core services include student management, tutor management, and study group coordination services. These services operate independently, allowing individual components to be updated or scaled based on demand without requiring full system redeployment.

Containerization using Docker ensures that each microservice runs within a consistent and isolated runtime environment. Docker images were built during the CI/CD process and stored

in Amazon Elastic Container Registry (ECR), enabling secure and version-controlled image management.

Table 1: ***: Core Microservices and Their Responsibilities***

Service Name	**Primary Responsibility**	**Technology Stack**	**Deployment Platform**
Student Service	Manages student profiles, authentication linkage, and participation in study groups	Node.js, Express	Amazon ECS
Tutor Service	Handles tutor registration, availability, and tutoring session management	Node.js, Express	Amazon ECS
Study Group Service	Coordinates study group creation, scheduling, and membership	Node.js, Express	Amazon ECS
User/Auth Service	Manages user identity integration and role-based access	Node.js, JWT	Amazon ECS

4.2 AWS Infrastructure Overview

The system was deployed within a dedicated Amazon Virtual Private Cloud (VPC), providing logical isolation and controlled network access. The VPC was segmented into public and private subnets across multiple availability zones to enhance availability and fault tolerance.

An Application Load Balancer (ALB) was deployed within the public subnet to manage incoming HTTP traffic. The ALB distributes requests to ECS services running in private subnets, ensuring that application containers are not directly exposed to the internet. This design improves both security and scalability.

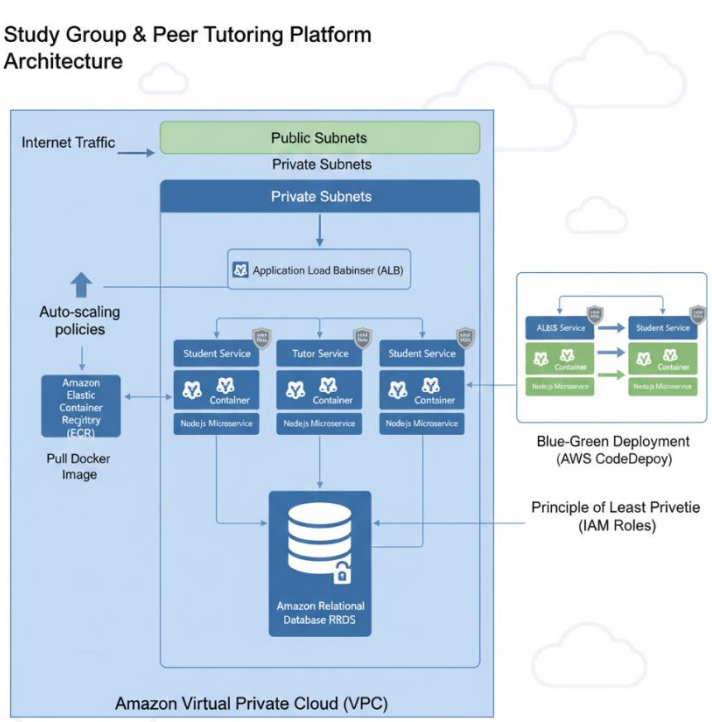

Figure 2: AWS Microservices Architecture of the Study Group & Peer Tutoring Platform

VPC Configuration: A dedicated VPC was created to house all resources.

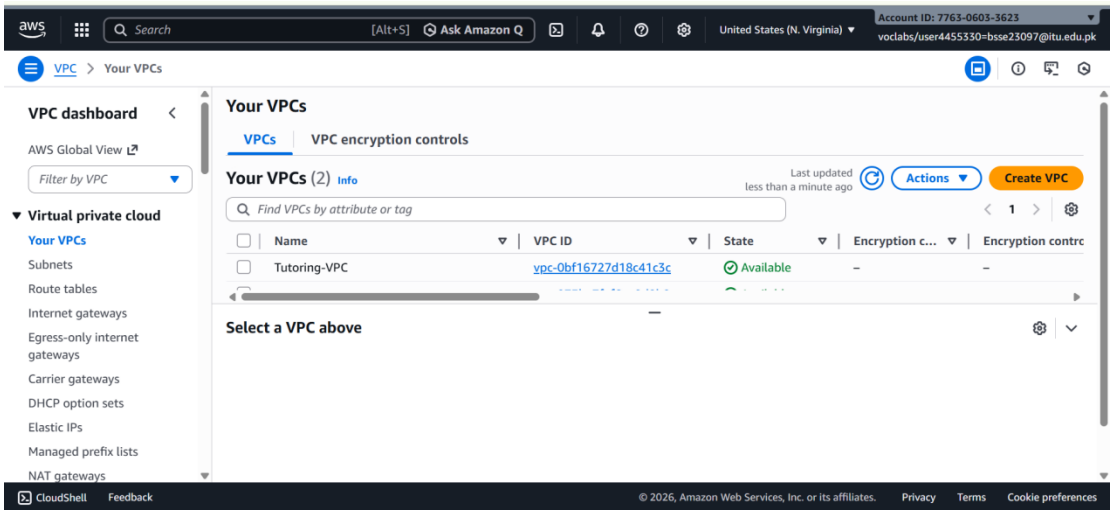

Figure 3: Configuration of the Virtual Private Cloud (VPC) for Network Isolation.

DNS & Connectivity: To ensure seamless communication between microservices, DNS Hostnames and DNS Support were enabled within the VPC settings.

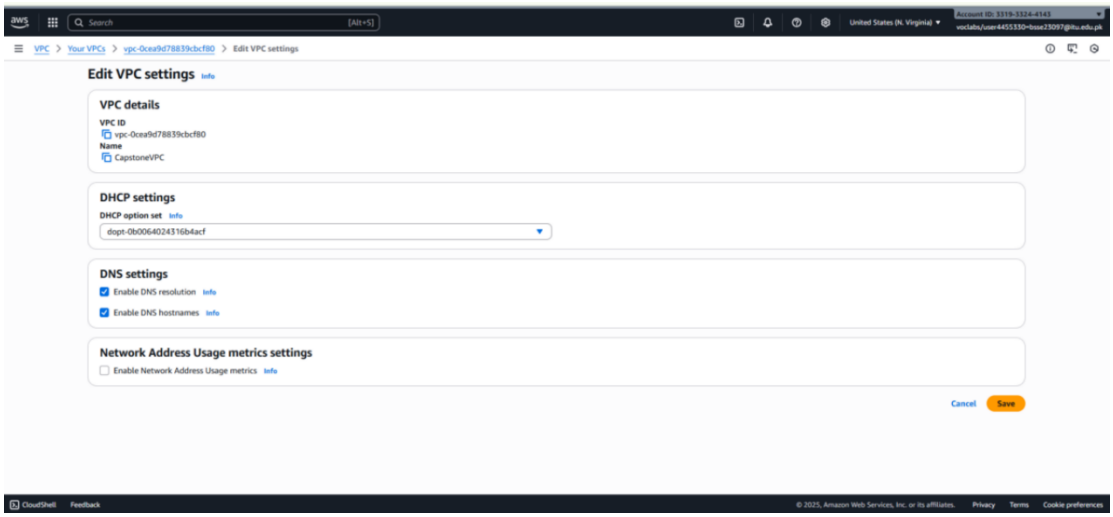

Figure 4: Enabling DNS Hostnames within the VPC Attributes.

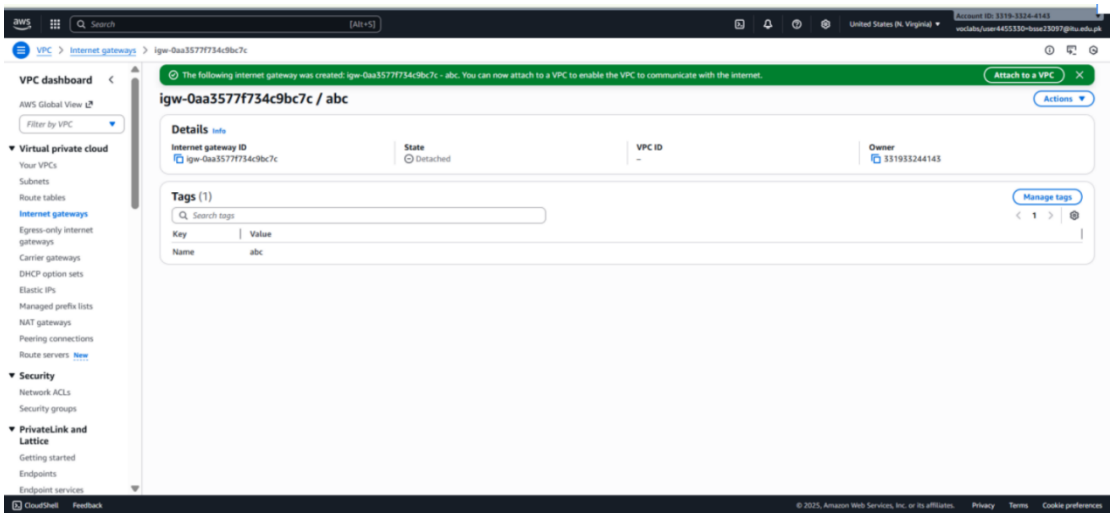

Figure 5 Confirmation of DNS Resolution Support for Internal Service Discovery.

Subnet Strategy: We implemented a multi-AZ (Availability Zone) strategy using 4 subnets—two public subnets for the Application Load Balancer (ALB) and two private subnets for the ECS Tasks and RDS Database to prevent direct public exposure.

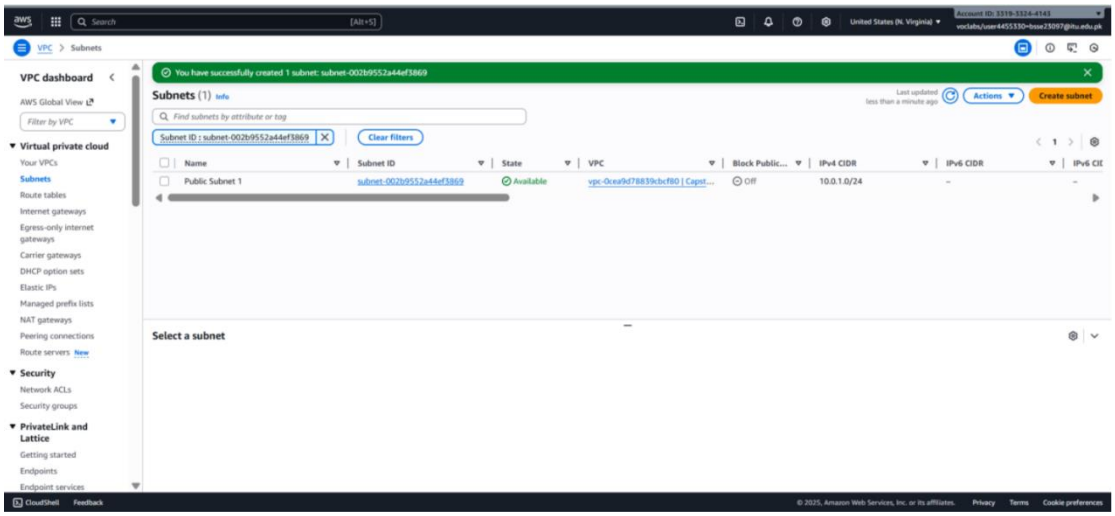

Figure 6: Subnet Architecture: Allocation of Public and Private Subnets across Availability Zones.

4.3 Container Orchestration with Amazon ECS

Amazon Elastic Container Service (ECS) was selected as the container orchestration platform due to its tight integration with AWS services and simplified operational management. ECS services were configured to run Docker containers using task definitions that specify CPU, memory, networking, and IAM permissions.

Auto-scaling policies were applied to ECS services to dynamically adjust the number of running tasks based on traffic patterns. This ensures optimal performance during peak academic periods while maintaining cost efficiency during low-usage intervals.

4.4 Database Architecture

Persistent data storage was implemented using Amazon Relational Database Service (RDS). The database instance was deployed within a private subnet, ensuring that it remains inaccessible from the public internet. Access to the database is strictly controlled through security groups and IAM policies.

This architecture supports data consistency, transactional integrity, and automated backups, making it suitable for handling academic records, scheduling data, and user profiles.

The backend relies on Amazon RDS for persistence. We initialized the schema to handle user authentication, tutor request tracking, and study group management.

- **Schema Implementation:** Tables were created to store session data and student-tutor mappings.

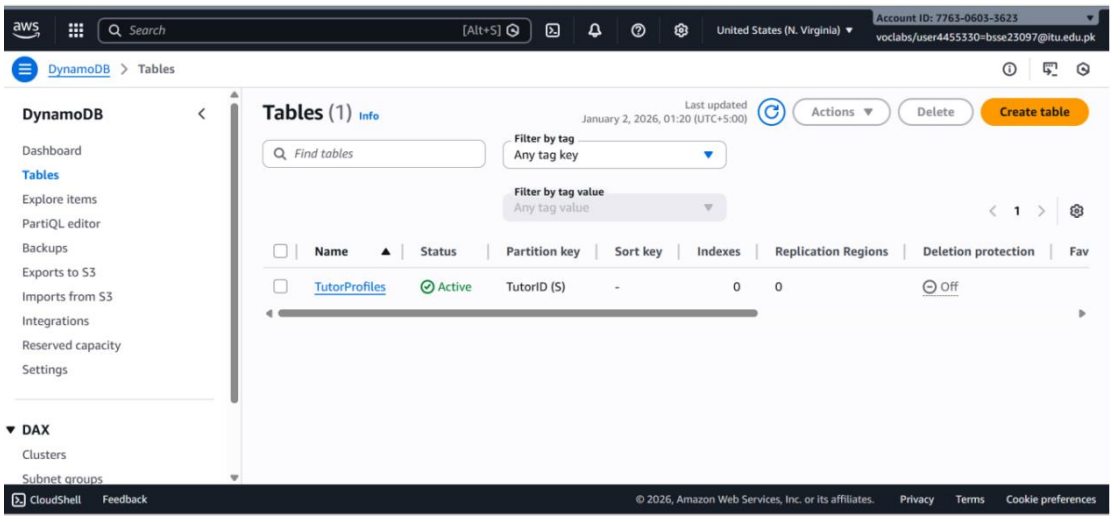

Figure 7: Initialization of Database Schema and Table Structures in Amazon RDS

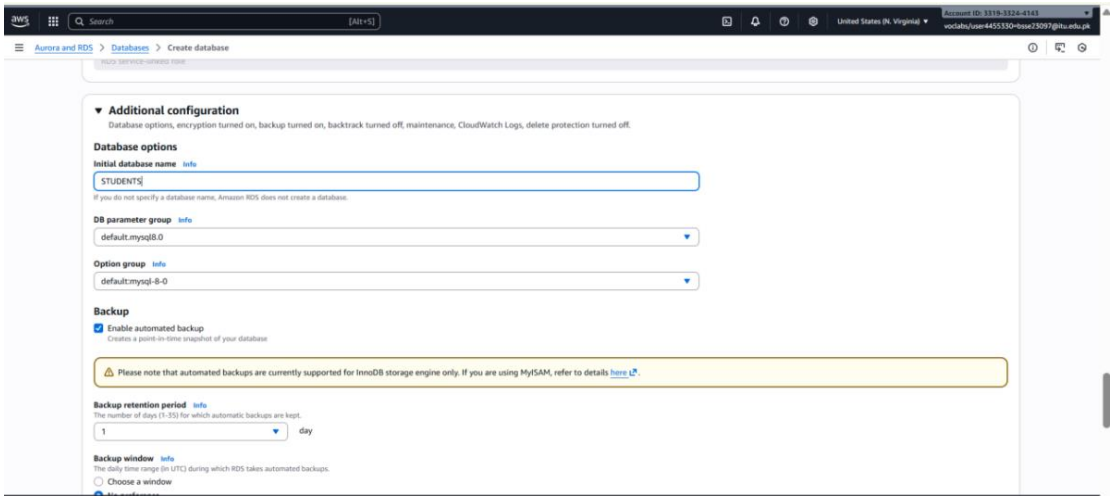

Figure 8: Database Connectivity and SQL Query Execution Verification.

Table 2: AWS Services Utilized in the Platform

AWS Service	Purpose	Justification
Amazon ECS	Container orchestration	Simplifies management and scaling of Dockerized microservices
Amazon ECR	Container image registry	Secure and version-controlled image storage
Application Load Balancer	Traffic distribution	Enables load balancing and Blue-Green deployments
Amazon RDS	Relational database	Managed, reliable, and secure persistent storage

Amazon VPC	Network isolation	Provides secure logical separation of resources
AWS CodePipeline	CI/CD orchestration	Automates build and deployment workflow
AWS CodeBuild	Build automation	Executes builds and Docker image creation
AWS CodeDeploy	Deployment automation	Enables Blue-Green and rolling deployments

5. Implementation Details

5.1 Service Development

All microservices were developed using Node.js, leveraging its non-blocking I/O model and suitability for scalable web applications. Each service follows a modular code structure, separating routing, business logic, and data access layers to improve maintainability and testability.

Environment-specific configurations were externalized using environment variables, enabling seamless deployment across different environments without modifying application code.

Deployment & Orchestration

We utilized **Amazon ECS (Elastic Container Service)** with Fargate (serverless) to manage our containers.

- **ECS Cluster:** A cluster was defined to manage the frontend and backend services.
- **Service & Task Definitions:** Tasks were configured with 0.5 vCPU and 1GB RAM to optimize costs while maintaining performance.

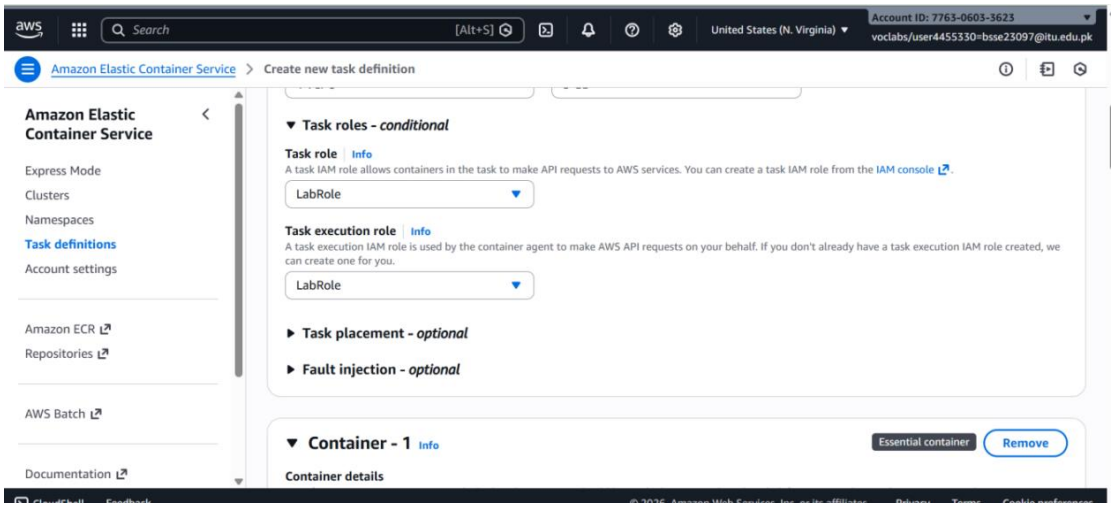

Figure 9 :Amazon ECS Cluster Initialization and Infrastructure Selection (Fargate).

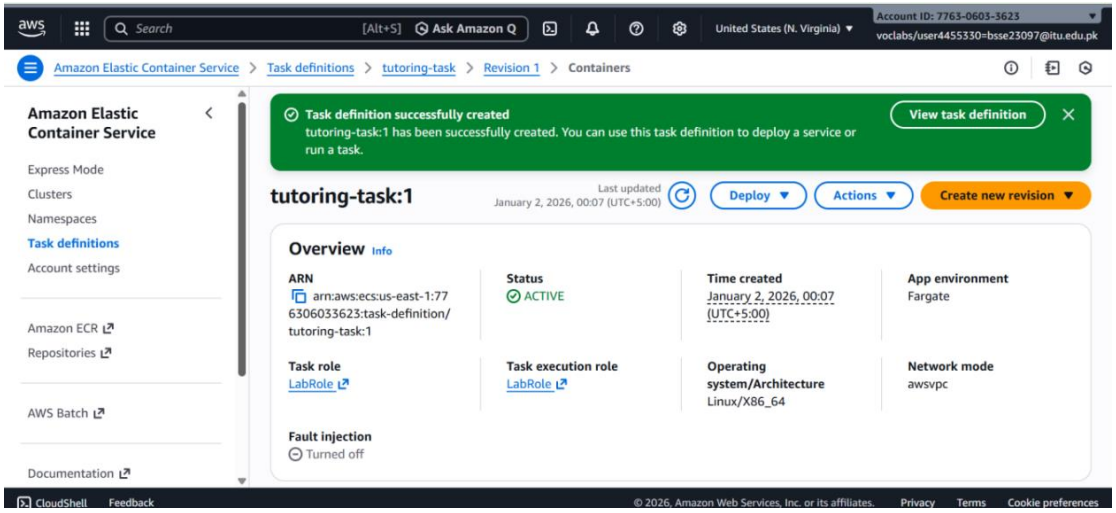

Figure 10: Task created

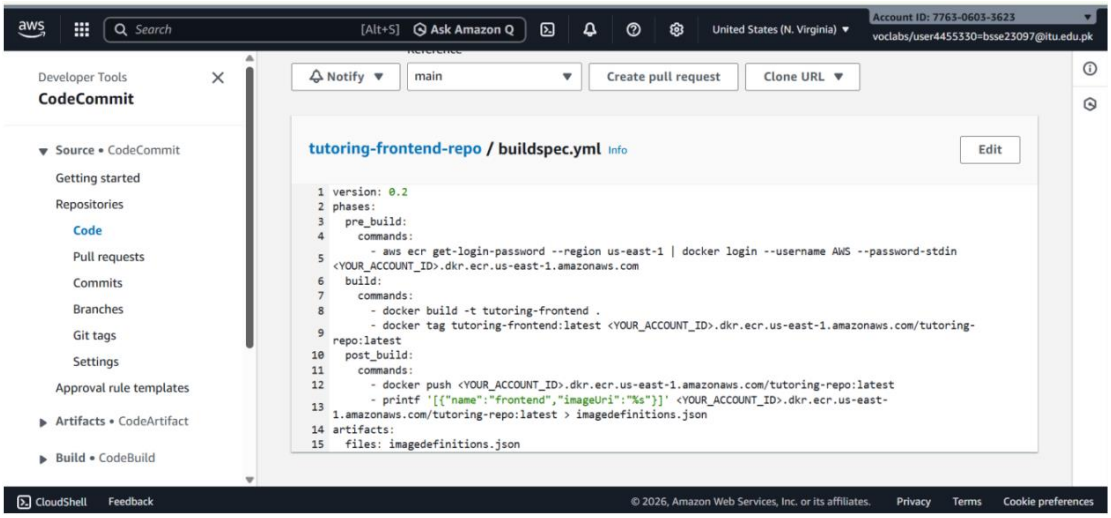

Figure 11 : Definition of Task Parameters (vCPU and Memory Allocation).

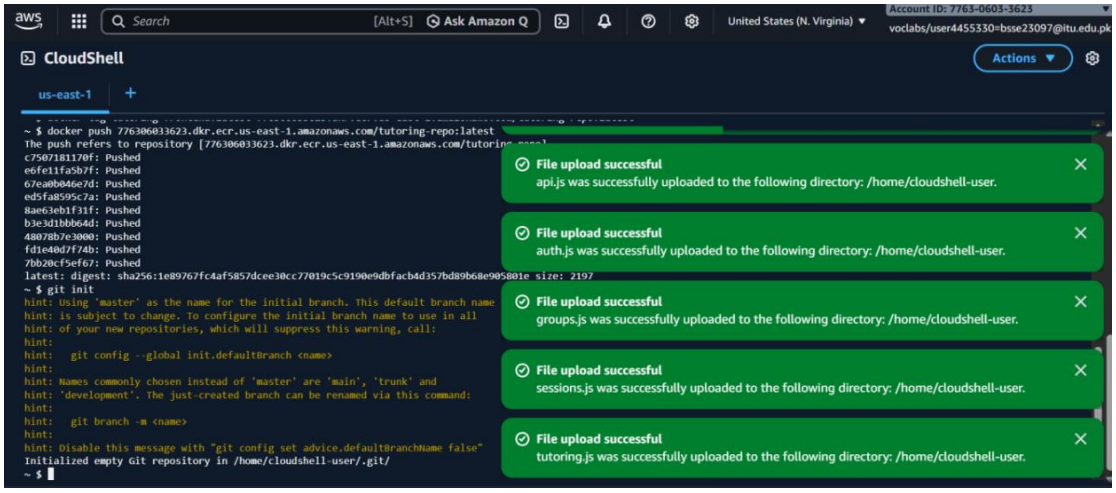

Figure 12: Configuration of Container Definitions and Port Mappings.

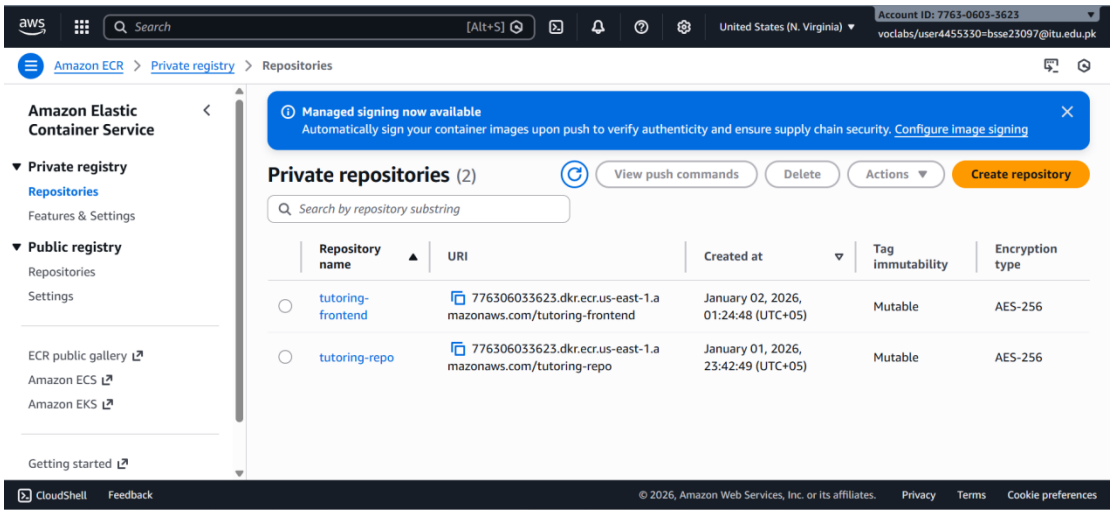

Figure 13: Deployment of ECS Service and Initial Task Running Status.

5.2 Containerization Strategy

Docker was used to containerize each microservice. Individual Dockerfiles were created for each service, defining the base image, dependency installation, application build steps, and runtime configuration. This approach ensures reproducibility and eliminates environment inconsistencies.

Docker images were automatically built during the CI/CD pipeline and tagged using version identifiers before being pushed to Amazon ECR.

Containerization & Registry

The application code was containerized using Docker to ensure environment parity.

- **ECR Repository:** We created a private repository named tutoring-repo to store our production-ready images.

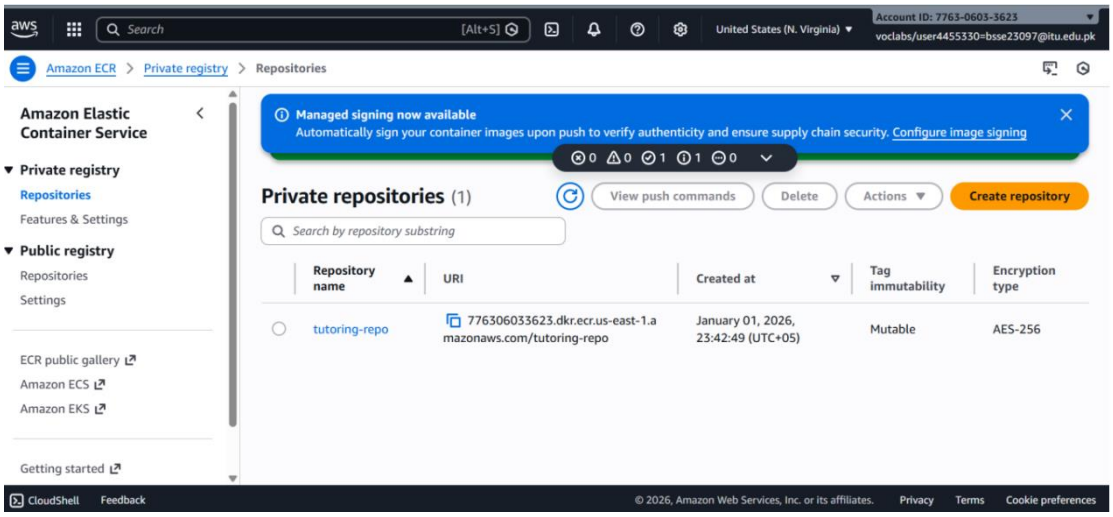

Figure 14: Amazon Elastic Container Registry (ECR) Private Repository Setup.

- **Build Process:** Using the buildspec.yml, images were tagged and pushed to ECR.

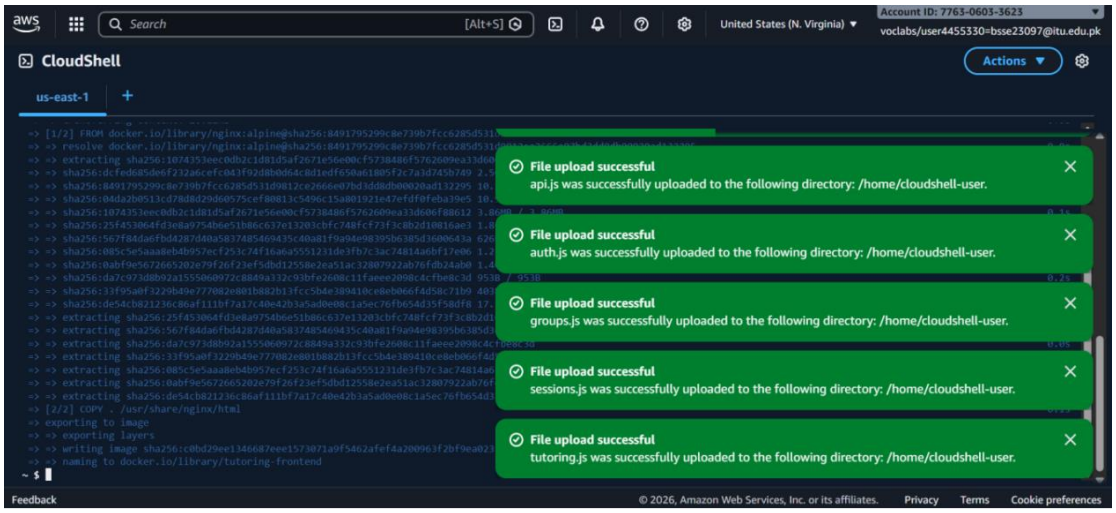

Figure 15: Execution of Docker Build Command via AWS CodeBuild.

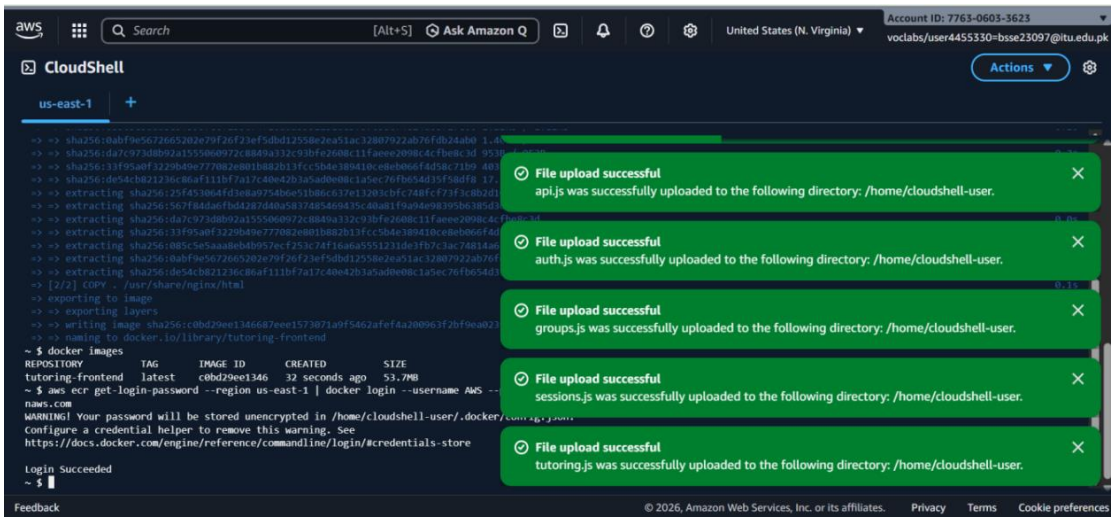

Figure 16: Container Image Tagging and Versioning for Registry Submission.

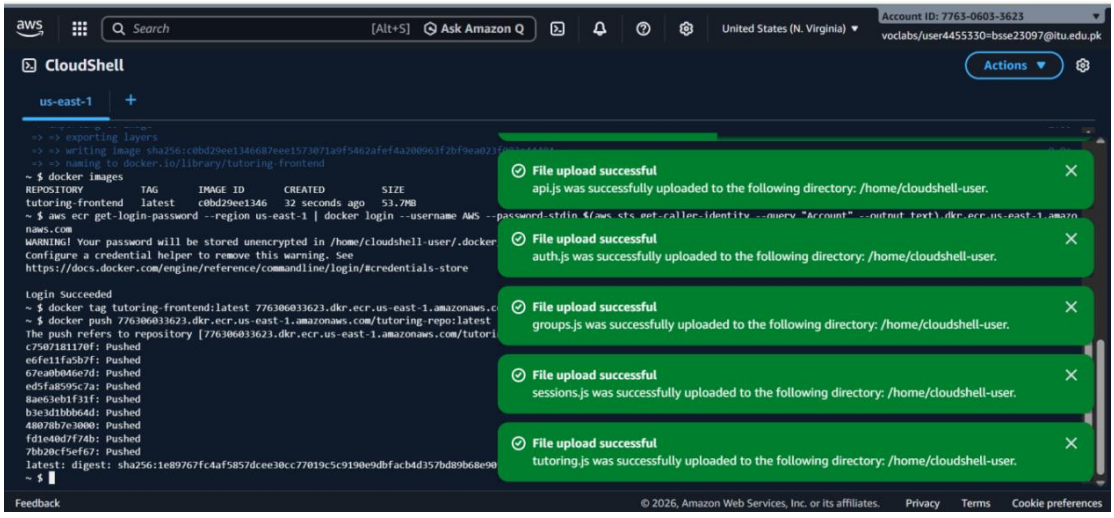

Figure 17 Successful Push of Frontend/Backend Images to ECR.

6. CI/CD Pipeline Design and Automation

To achieve continuous integration and continuous deployment, a fully automated CI/CD pipeline was implemented using AWS-native services integrated with GitHub.

Table 3: CI/CD Pipeline Stages and Responsibilities

Pipeline Stage	Tool Used	Description
Source	GitHub	Hosts application source code and triggers pipeline on commits

Build	AWS CodeBuild	Runs tests, builds Docker images, and pushes images to ECR
Image Storage	Amazon ECR	Stores versioned Docker images securely
Deploy	AWS CodeDeploy	Manages Blue-Green and rolling ECS deployments

To achieve zero-downtime deployments, we implemented a Blue-Green strategy via AWS CodeDeploy.

- **Target Groups:** Two target groups were utilized:
    1. **TG-Primary (Blue):** Currently serving live traffic.
    2. **TG-Replacement (Green):** Where the new version is tested before the swap.

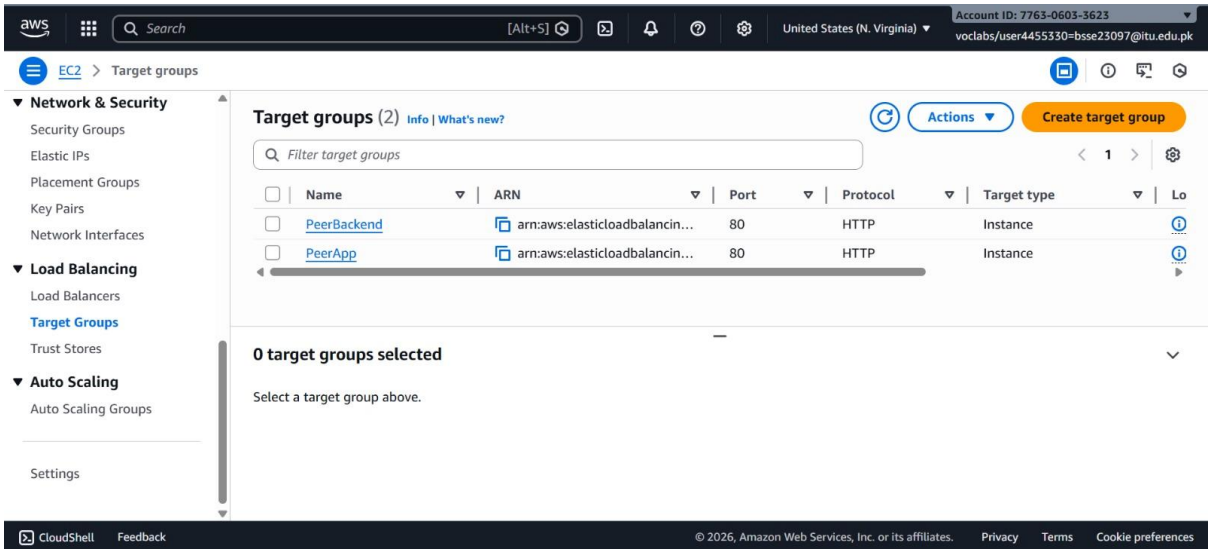

Figure 18: Target Groups

6.1 Pipeline Workflow

The CI/CD pipeline begins with a source code commit to the GitHub repository. Upon detecting changes, AWS CodePipeline automatically triggers the build and deployment process.

AWS CodeBuild executes the build phase using a defined buildspec.yml file. This phase includes dependency installation, automated testing, Docker image creation, and image push to Amazon ECR.

AWS CodeDeploy manages the deployment stage, orchestrating Blue-Green deployments for ECS services. This deployment strategy allows the new version of the application to be deployed alongside the existing version, with traffic gradually shifted after successful health checks.

Figure 2 presents the CI/CD pipeline flow, illustrating the integration between GitHub, CodePipeline, CodeBuild, ECR, and CodeDeploy.

6.2 Blue-Green Deployment Strategy

Blue-Green deployment was implemented to eliminate service downtime during application updates. In this approach, the existing production environment (Blue) continues serving traffic while a new environment (Green) is deployed and validated.

Once the Green environment passes health checks, traffic is redirected through the Application Load Balancer. If any issues are detected, the system can instantly revert to the Blue environment, minimizing operational risk.

Table 4: ***Implemented Security Measures***

Security Area	Implementation	Purpose
Identity Management	IAM roles for ECS tasks	Prevents hard-coded credentials and enforces least privilege
Network Security	VPC with public/private subnets	Isolates sensitive components from public access
Traffic Control	Security Groups	Restricts inbound and outbound communication
Secrets Management	AWS Secrets Manager & environment variables	Protects sensitive configuration data
Data Protection	RDS in private subnet	Prevents direct internet exposure of database

7. Security and Networking

Security was a core design consideration throughout the system architecture. The platform adheres to the principle of least privilege and AWS security best practices.

7.1 Identity and Access Management

IAM roles were assigned to ECS tasks, granting only the permissions required to access services such as ECR, CloudWatch, and RDS. This approach eliminates the need for hard-coded credentials within application containers.

IAM Roles: An ECS Task Execution Role was associated with the cluster to grant permissions for pulling images from ECR and sending logs to CloudWatch.

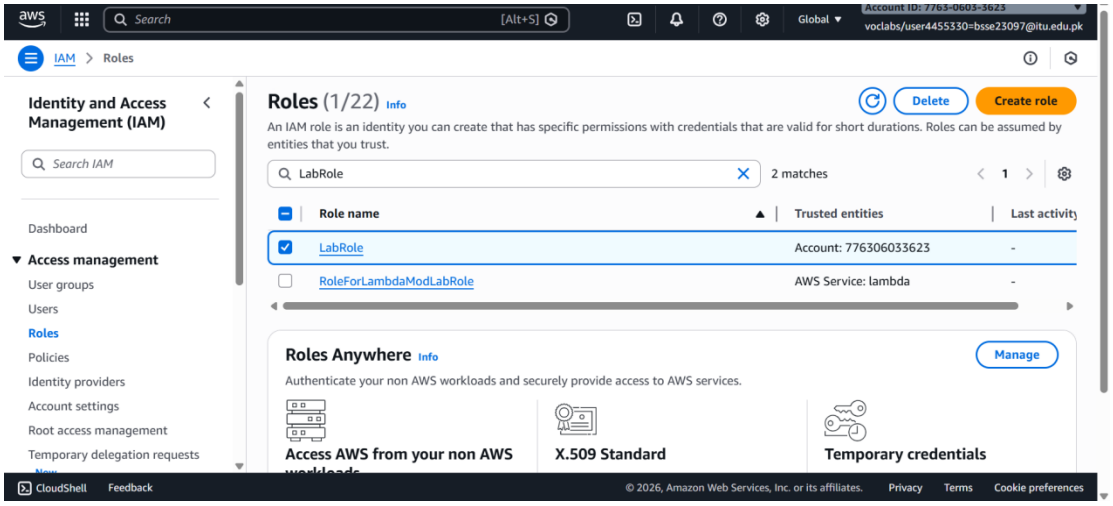

Figure 19: IAM Role Customization and Association for ECS Task Execution.

Security Groups: * ALB Security Group: Permits inbound traffic on Ports 80 (HTTP) and 443 (HTTPS).

- **App Security Group:** Only allows traffic originating from the ALB.
- **DB Security Group:** Restricted to Port 3306 (MySQL), allowing access only from the App Security Group.

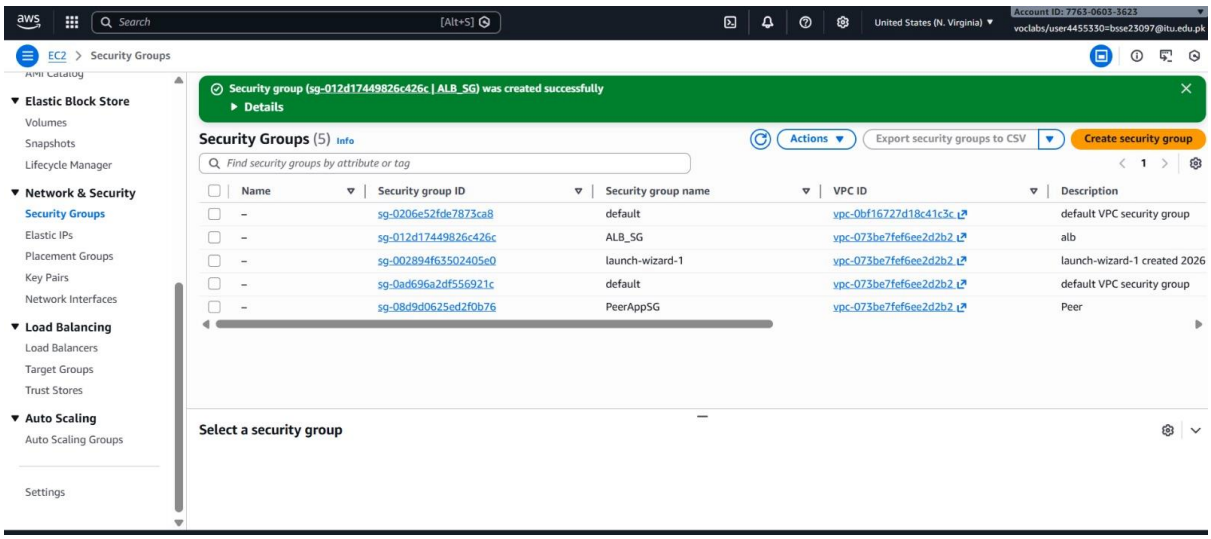

Figure 20: Security Groups

7.2 Performance Monitoring & Testing

Post-deployment, the system was subjected to rigorous monitoring and load testing.

- **ALB Load Testing:** We conducted load tests to verify the Application Load Balancer's ability to distribute traffic across multiple targets. This confirmed that the "Healthy Host Count" remained stable under stress.

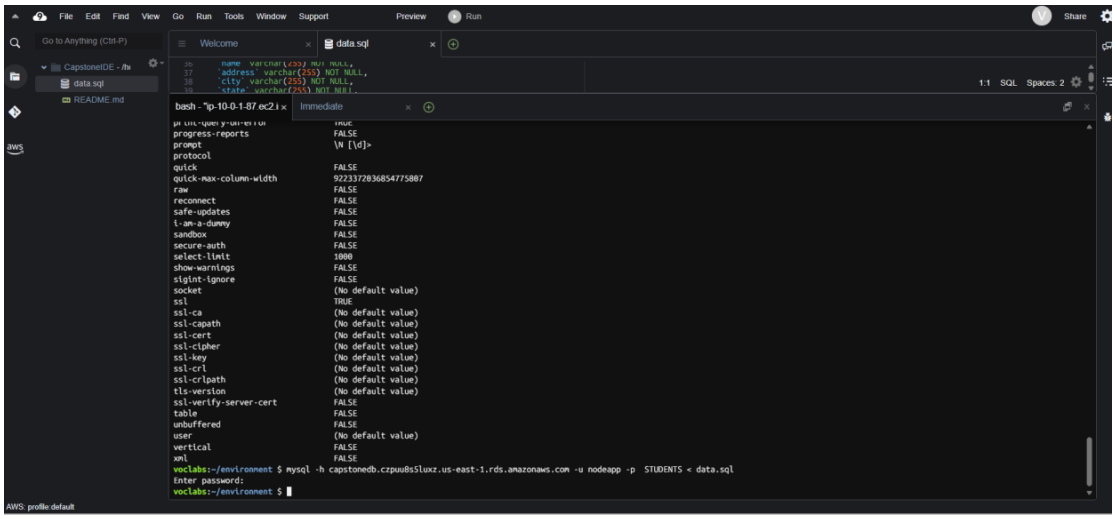

Figure 21: Application Load Balancer (ALB) Configuration and Listener Rules.

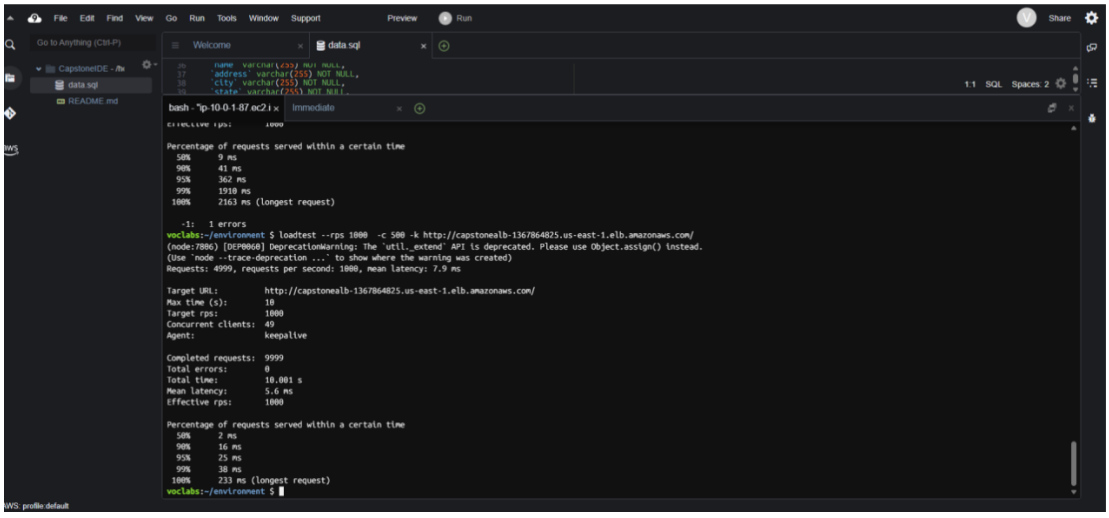

Figure 222: Stress Testing: Simulating Concurrent User Traffic to the ALB.

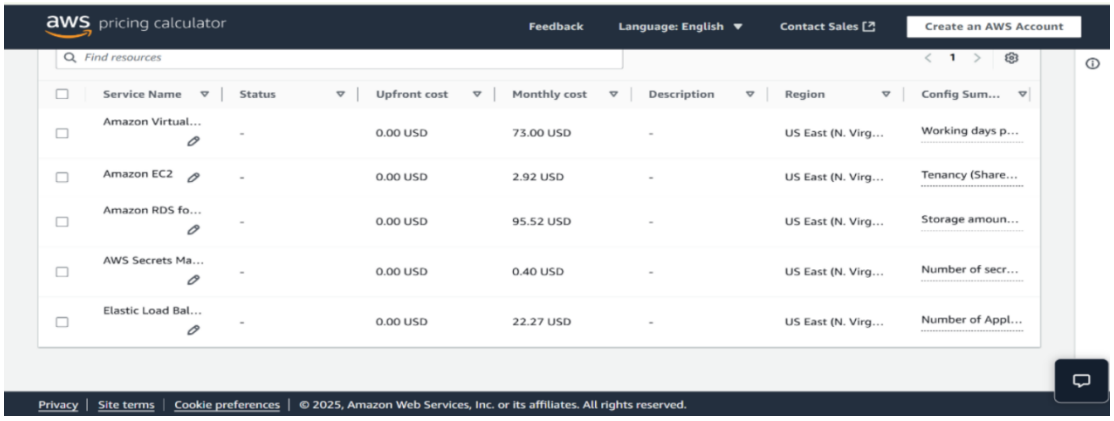

Figure 23: Load Balancer Latency and Request Count Analysis.

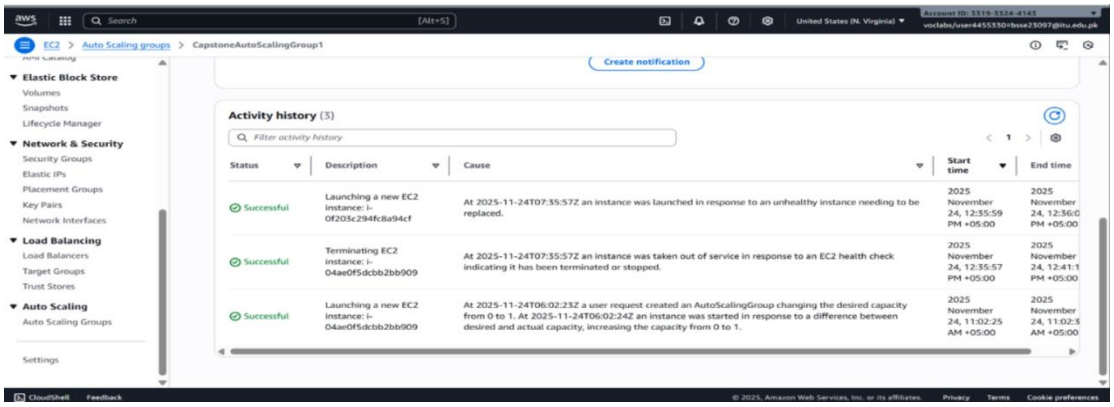

Figure 24: Successful Response Validation under Peak Load Conditions.

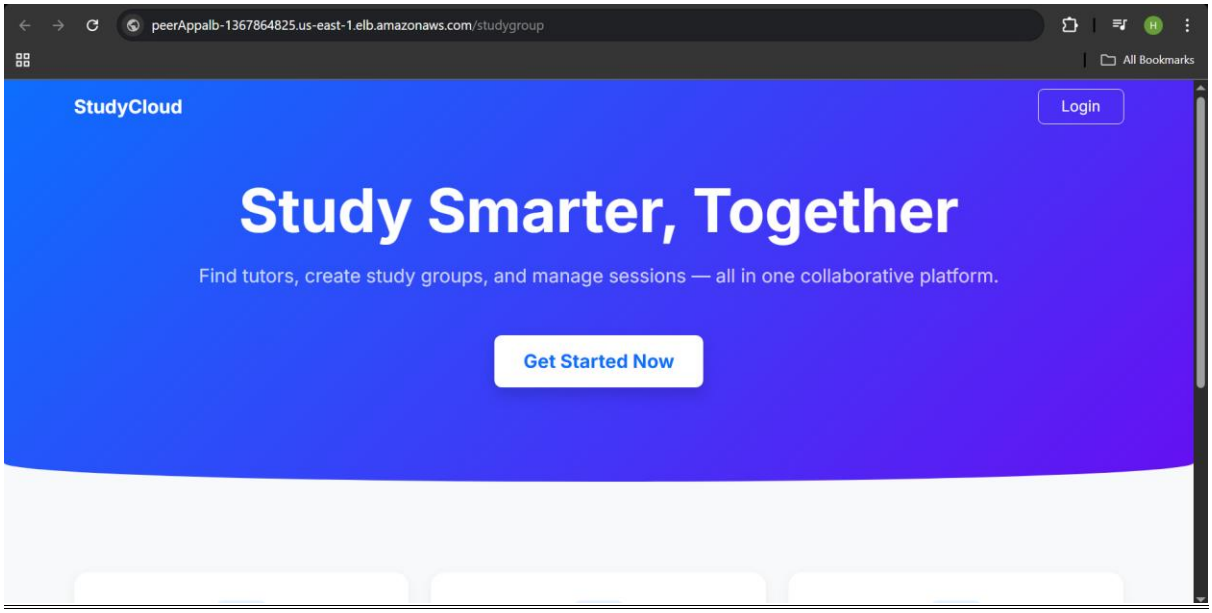

Figure 25 : Final System Health Check and Blue-Green Deployment Completion with ALB

8. Network Security

Security groups were configured to restrict inbound and outbound traffic between components. Only the ALB is publicly accessible, while ECS tasks and the RDS database reside in private subnets.

Sensitive configuration values and credentials were managed securely using environment variables and AWS Secrets Manager, ensuring that confidential information is not exposed in source code repositories.

The deployed platform demonstrated high availability, scalability, and operational stability. Automated deployments significantly reduced manual intervention and deployment errors. Blue-Green deployment ensured uninterrupted service availability during updates.

The microservices architecture enabled independent scaling of services, allowing the system to handle varying workloads efficiently. Monitoring and logging mechanisms facilitated rapid issue detection and resolution.

9. Limitations and Future Work

While the system meets its primary objectives, several enhancements can be considered for future development. These include the integration of an API Gateway for enhanced request management, advanced monitoring dashboards, and the adoption of infrastructure-as-code tools such as AWS CloudFormation or Terraform.

Future iterations may also incorporate enhanced security features, such as fine-grained access control and improved observability using distributed tracing.

10. Conclusion

This project successfully demonstrated the design and deployment of a cloud-native Study Group & Peer Tutoring Platform using microservices and DevOps practices on AWS. By leveraging containerization, automated CI/CD pipelines, and Blue-Green deployment strategies, the system achieves high scalability, security, and reliability. The project validates the effectiveness of modern cloud architectures in addressing real-world academic collaboration challenges and provides a strong foundation for future expansion.

References

[1] Amazon Web Services, "Amazon Elastic Container Service Documentation," AWS, 2024.

[2] Amazon Web Services, "AWS CodePipeline User Guide," AWS, 2024.

[3] Amazon Web Services, "Blue-Green Deployments with AWS CodeDeploy," AWS, 2024.

[4] Newman, S., *Building Microservices*, O'Reilly Media, 2021.