



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.

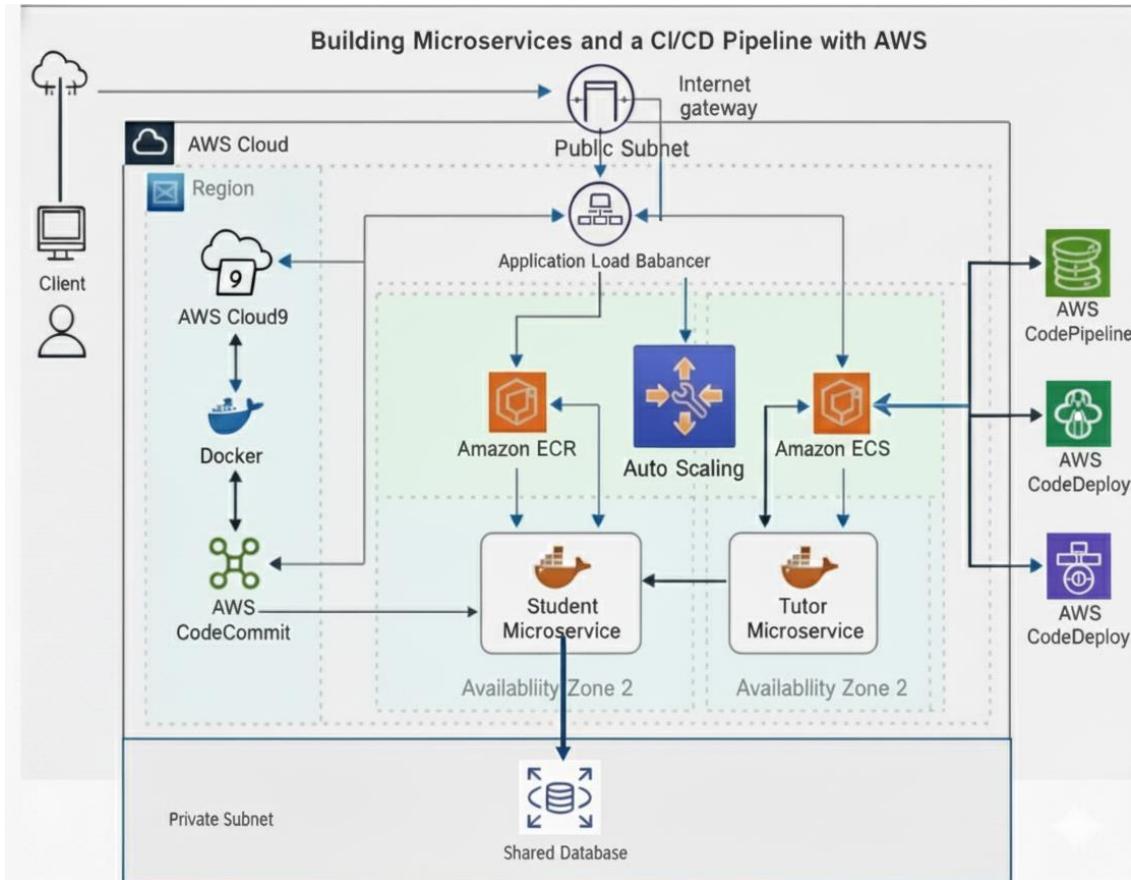


Figure 1: Architecture Diagram

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.

Study Group & Peer Tutoring Platform Architecture

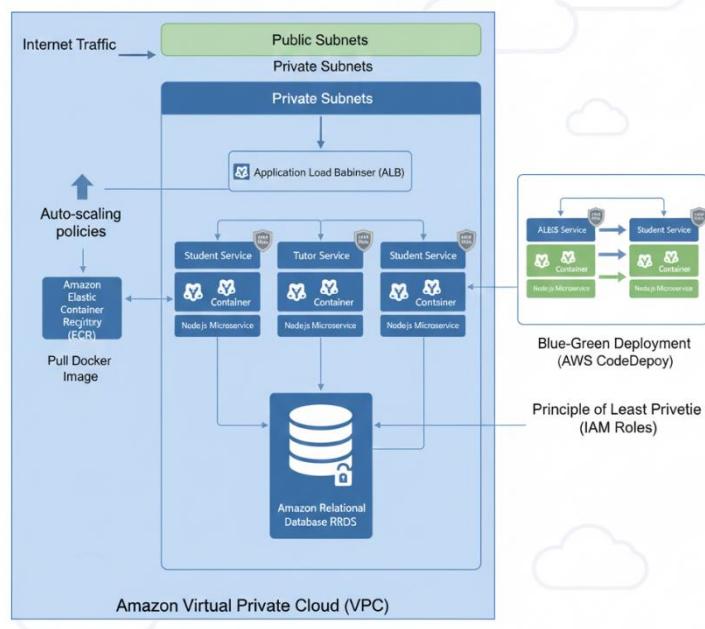


Figure 2: AWS Microservices Architecture of the Study Group & Peer Tutoring Platform

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

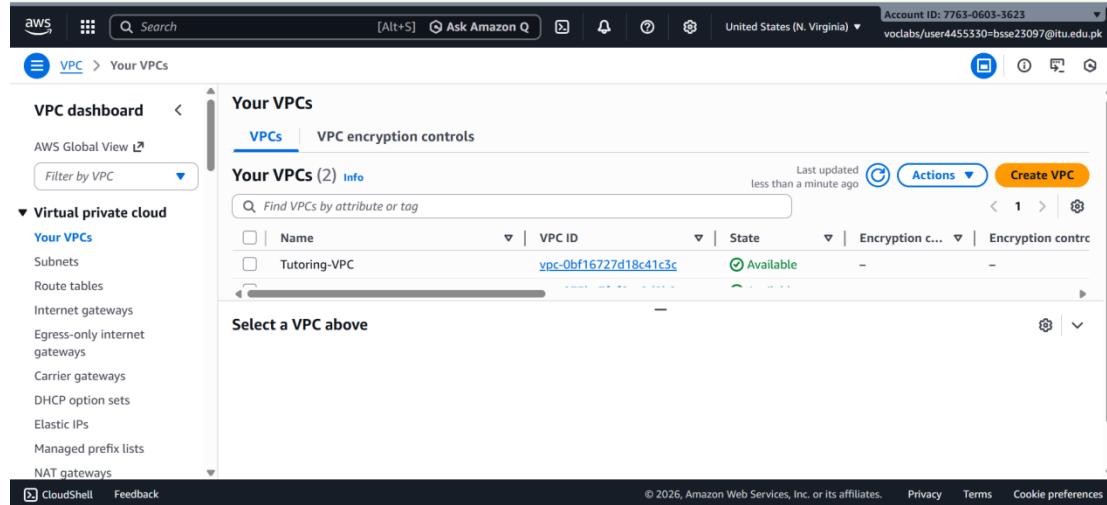


Figure 3: Configuration of the Virtual Private Cloud (VPC) for Network Isolation.

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

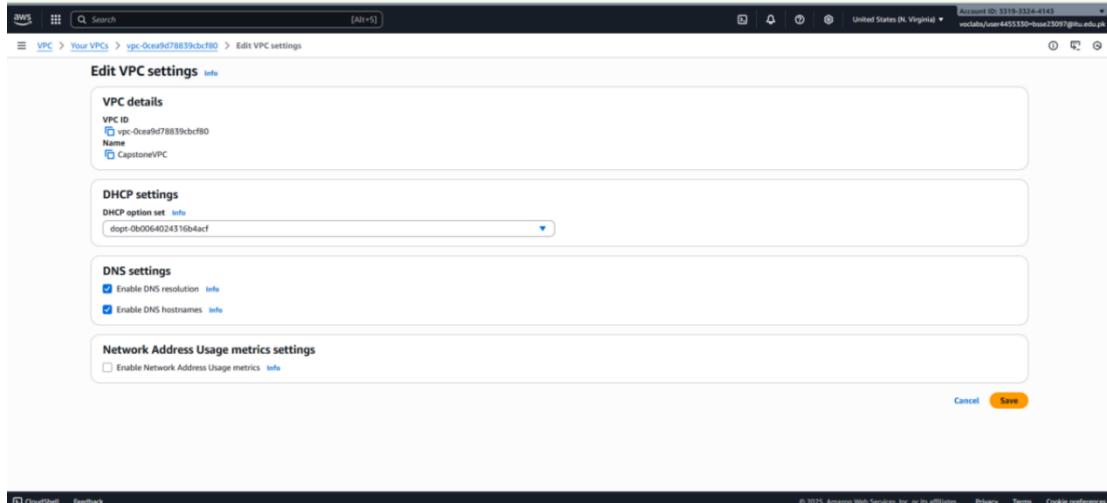


Figure 4: Enabling DNS Hostnames within the VPC Attributes.

□

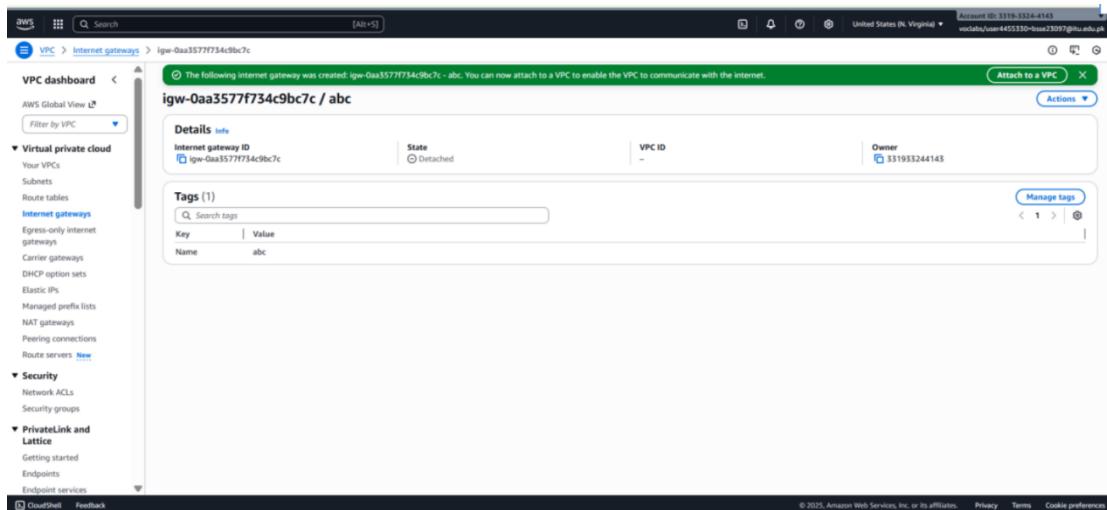


Figure 5 Confirmation of DNS Resolution Support for Internal Service Discovery.

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.

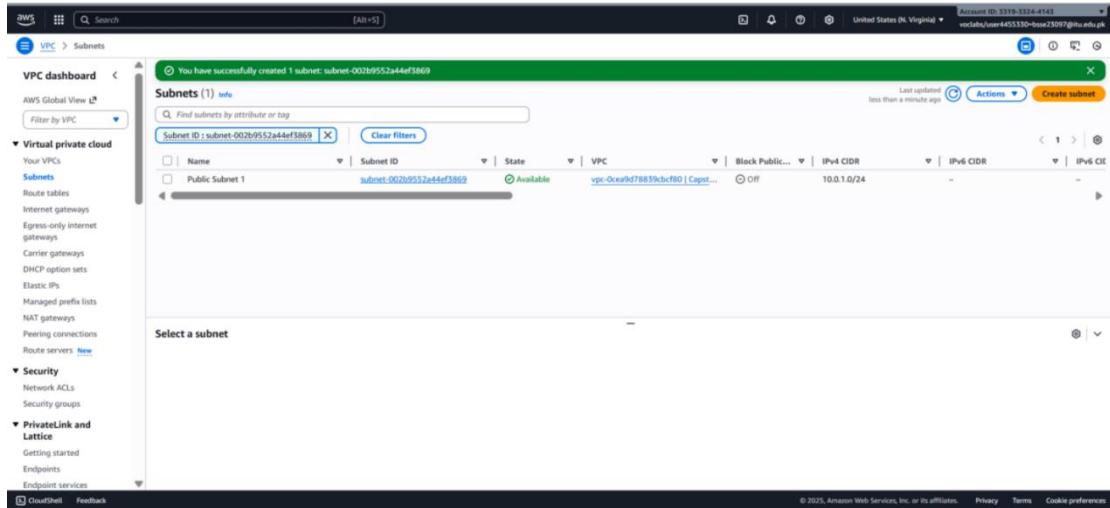


Figure 6: Subnet Architecture: Allocation of Public and Private Subnets across Availability Zones.

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.

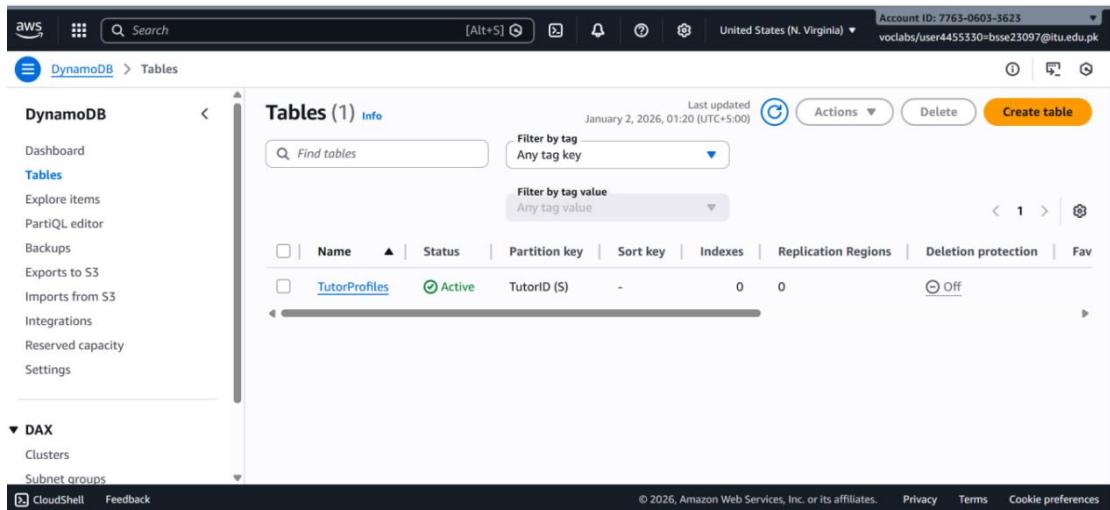


Figure 7: Initialization of Database Schema and Table Structures in Amazon RDS

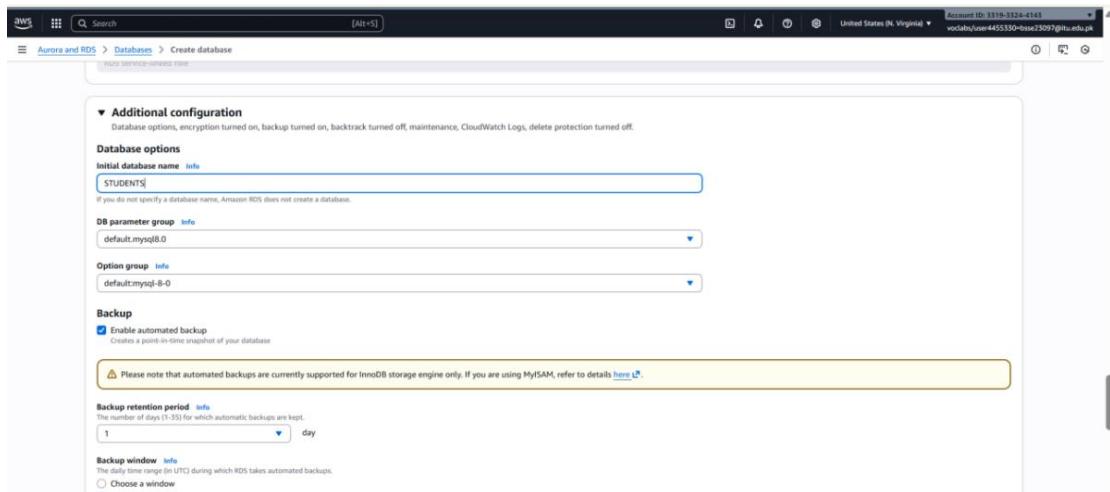


Figure 8: Database Connectivity and SQL Query Execution Verification.

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.

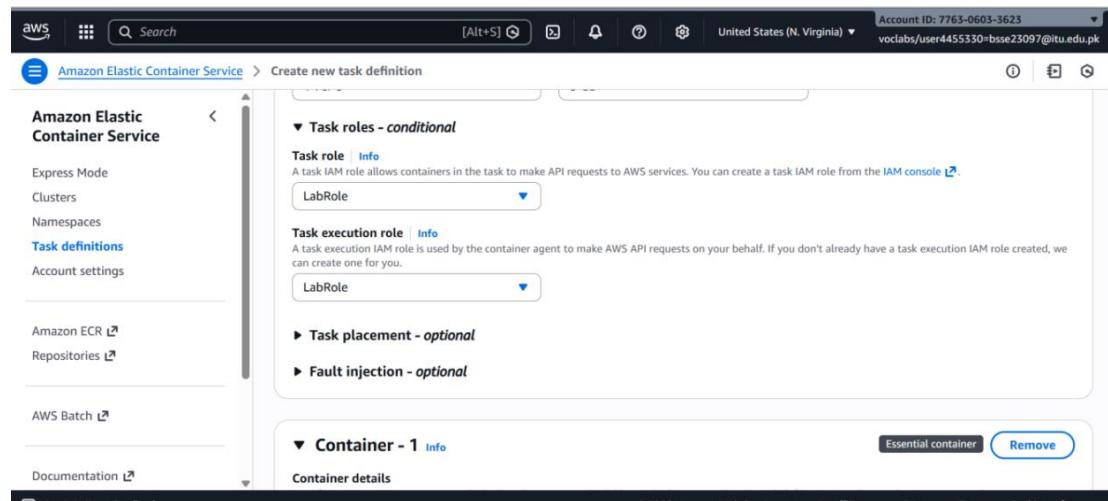


Figure 9 :Amazon ECS Cluster Initialization and Infrastructure Selection (Fargate).

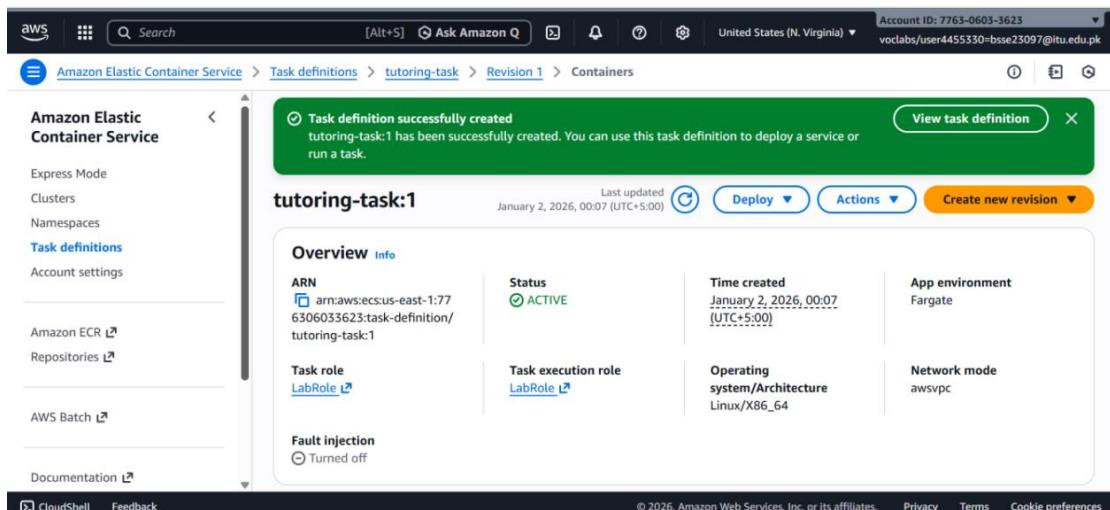


Figure 10: Task created

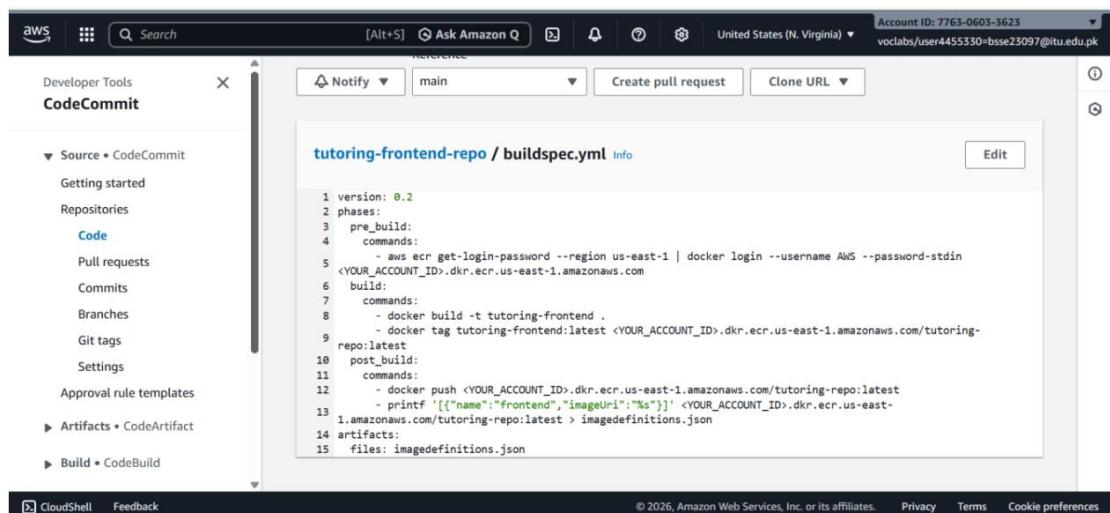


Figure 11 : Definition of Task Parameters (vCPU and Memory Allocation).

```

aws CloudShell
us-east-1 + 1

~ $ docker push 776306033623.dkr.ecr.us-east-1.amazonaws.com/tutoring-repo:latest
The push refers to repository [776306033623.dkr.ecr.us-east-1.amazonaws.com/tutoring-repo]
c750f181170f: Pushed
e6fe11fa5b7f: Pushed
67ea0b046e7d: Pushed
ed5ta895c7a: Pushed
8ae63eb1f31f: Pushed
b3e3d1bbb64d: Pushed
48078b7e3000: Pushed
fd1e40d7f74b: Pushed
7b028c15ef6f: Pushed
lambda-layer: sha256:1e89767fc4af5857dce30cc77019c5c9190e9dbfacb4d357bd89b68e905801e size: 2197
~ $ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
hint:
hint: Disable this message with "git config set advice.defaultBranchName false"
Initialized empty Git repository in /home/cloudshell-user/.git/
~ $ 

```

File upload successful
api.js was successfully uploaded to the following directory: /home/cloudshell-user.

File upload successful
auth.js was successfully uploaded to the following directory: /home/cloudshell-user.

File upload successful
groups.js was successfully uploaded to the following directory: /home/cloudshell-user.

File upload successful
sessions.js was successfully uploaded to the following directory: /home/cloudshell-user.

File upload successful
tutoring.js was successfully uploaded to the following directory: /home/cloudshell-user.

Figure 12: Configuration of Container Definitions and Port Mappings.

Amazon Elastic Container Service

Private registry

Repositories

Features & Settings

Public registry

Repositories

Settings

ECR public gallery

Amazon ECS

Amazon EKS

Getting started

Amazon ECR > Private registry > Repositories

Managed signing now available

Automatically sign your container images upon push to verify authenticity and ensure supply chain security. [Configure image signing](#)

Private repositories (2)

Repository name	URI	Created at	Tag immutability	Encryption type
tutoring-frontend	776306033623.dkr.ecr.us-east-1.amazonaws.com/tutoring-frontend	January 02, 2026, 01:24:48 (UTC+05)	Mutable	AES-256
tutoring-repo	776306033623.dkr.ecr.us-east-1.amazonaws.com/tutoring-repo	January 01, 2026, 23:42:49 (UTC+05)	Mutable	AES-256

View push commands Delete Actions Create repository

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.

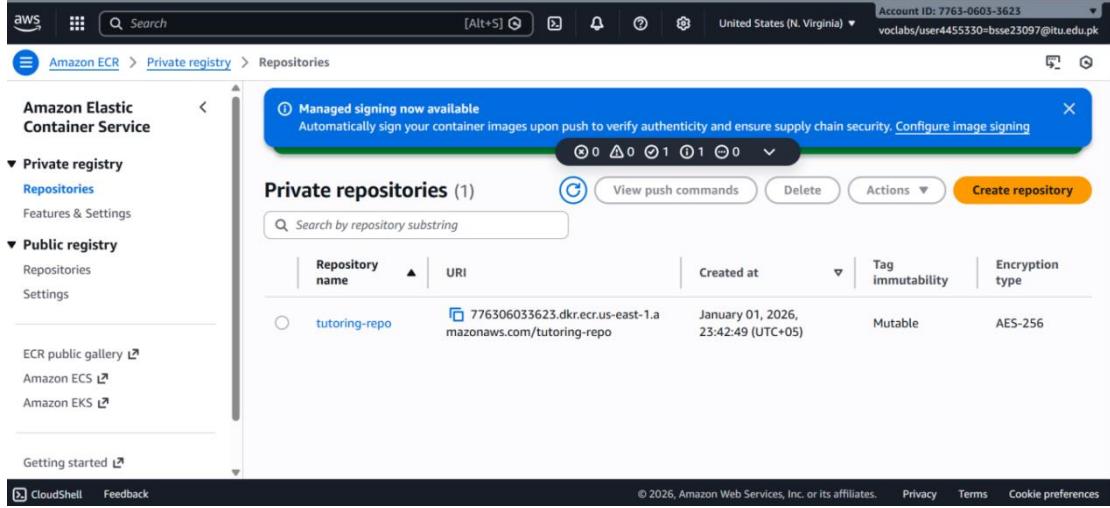


Figure 14: Amazon Elastic Container Registry (ECR) Private Repository Setup.

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

```

aws
[CloudShell]
us-east-1 + [1/2] FROM docker.io/library/nginx:alpine@sha256:8491795299c8e739b7fcc6285d531...
>>> >> resolve docker.io/library/nginx:alpine@sha256:8491795299c8e739b7fcc6285d531...
>>> >> extracting sha256:1074353ee0db2c1d1da2f2671e56e00cf5738486f5762609ea33d00...
>>> >> sha256:8491795299c8e739b7fcc6285d531...0d9...
>>> >> sha256:04042a09513a07848d00675780813c5496c15a0a01921e47ef0f0feba39e...
>>> >> sha256:107433160d02c1d8105af2671e56e00c5738486f762609ea33d00...
>>> >> sha256:75f4530e4fd3e8a9745046e51000c637e13201chfc748fcf7f1f3802d1051a0e3...
>>> >> sha256:56f784d9fb4287d4ba5837485469435c408a119a0498395b6385d3...
>>> >> sha256:613195a0a3229049e77082e880a8013fc5bda498410cebeb6e6fad5d4c7109...
>>> >> sha256:de4c0821236c80af111bf7a1740e42b3a5ad0eb08c1a0ec76fb04d35f5df0d...
>>> >> extracting sha256:25f4530e4fd3e8a9754be51b86c637e13203chfc748fcf7f1f3802d1051a0e3...
>>> >> extracting sha256:d47973db92a1550e097228849312c930fe2908011faeee209804c1b0e...
>>> >> extracting sha256:d47973db92a1550e097228849312c930fe2908011faeee209804c1b0e...
>>> >> extracting sha256:0895e5aa1eb0957ecef253c747f16a6a5591231de1fb7c1ac7481a4...
>>> >> extracting sha256:0abf9e5672665202e79f26f23ef0d12558e8ea1ac2807922a0ab01...
>>> >> extracting sha256:de54cb821236c86a111bf7a1740e42b3a5ad0e08c1a5ec76fb04d35f5df0d...
>>> [2/2] COPY . /home/share/nginx
>>> exporting to image
>>> exporting layers
>>> writing image sha256:@bd29ee1346687eee1573071a9f5462afe4a200963f2fb9fea023...
>>> naming to docker.io/library/tutoring-frontend
~ 5
Feedback

```

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

The screenshot shows the AWS CloudShell interface. At the top, there's a search bar, an 'Ask Amazon Q' button, and a 'United States (N. Virginia)' dropdown. Below the header, the title 'CloudShell' is displayed, followed by 'us-east-1'. On the right side, there are 'Actions' and a gear icon.

The main area contains several terminal sessions:

- A terminal window showing the output of a series of curl commands to a local host port 8080, with the last command failing with a 503 error.
- A terminal window showing the upload of 'api.js' to the '/home/cloudshell-user' directory, indicated by a green success message.
- A terminal window showing the upload of 'auth.js' to the '/home/cloudshell-user' directory, indicated by a green success message.
- A terminal window showing the upload of 'groups.js' to the '/home/cloudshell-user' directory, indicated by a green success message.
- A terminal window showing the upload of 'sessions.js' to the '/home/cloudshell-user' directory, indicated by a green success message.
- A terminal window showing the upload of 'tutoring.js' to the '/home/cloudshell-user' directory, indicated by a green success message.
- A terminal window showing the command to list Docker images, which lists 'tutoring-frontend' as the latest image.
- A terminal window showing the command to pull the Docker image for 'tutoring-frontend'.
- A terminal window showing the command to run the Docker container for 'tutoring-frontend'.
- A terminal window showing the command to log in to the Docker container.
- A terminal window showing the command to run 'aws configure' to set up AWS credentials.
- A terminal window showing the command to run 'aws configure' again, with a note about password security.
- A terminal window showing the command to run 'curl' on port 8080.

At the bottom left, there's a 'Login Succeeded' message with a small profile icon. The footer includes links for 'Feedback', '© 2026, Amazon Web Services, Inc. or its affiliates.', 'Privacy', 'Terms', and 'Cookie preferences'.

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

The screenshot shows the AWS CloudShell interface. At the top, there's a search bar, an 'Ask Amazon Q' button, and account information for 'United States (N. Virginia)'. Below the header, the title 'CloudShell' is displayed. The main area contains a terminal window with several command-line operations:

- Uploading files: 'aws s3 cp ./* s3://cloudshell-user/tutoring-frontend' and 'aws s3 cp api.js s3://cloudshell-user/tutoring-frontend'.
- Running Docker images: '\$ docker images' lists 'tutoring-frontend' as the latest image.
- Logging in to Docker: '\$ aws ecr get-login-password --region us-east-1 | docker login --username AWS --password-stdin'.
- Configuring AWS CLI: 'Configure a credential helper to remove this warning. See https://docs.docker.com/engine/reference/commandline/login/#credentials-store'.
- Pushing Docker tags: '\$ docker tag tutoring-frontend:latest 776306033623.dkr.ecr.us-east-1.amazonaws.com/tutoring-frontend:latest' and '\$ docker push 776306033623.dkr.ecr.us-east-1.amazonaws.com/tutoring-frontend:latest'.
- Pushing other files: '\$ git push -u origin main' and '\$ git push -u origin develop'.

On the right side of the terminal, there are six green status cards, each indicating a successful file upload:

- 'File upload successful': 'api.js was successfully uploaded to the following directory: /home/cloudshell-user.'
- 'File upload successful': 'auth.js was successfully uploaded to the following directory: /home/cloudshell-user.'
- 'File upload successful': 'groups.js was successfully uploaded to the following directory: /home/cloudshell-user.'
- 'File upload successful': 'sessions.js was successfully uploaded to the following directory: /home/cloudshell-user.'
- 'File upload successful': 'tutoring.js was successfully uploaded to the following directory: /home/cloudshell-user.'

At the bottom left, there's a 'Feedback' link, and at the bottom right, there are links for 'Privacy', 'Terms', and 'Cookie preferences'.

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.

The screenshot shows the AWS CloudFront Target Groups page. The left sidebar has sections for Network & Security (Security Groups, Elastic IPs, Placement Groups, Key Pairs, Network Interfaces), Load Balancing (Load Balancers, Target Groups selected), Auto Scaling (Auto Scaling Groups), and Settings. The main content area is titled "Target groups (2)" and lists "PeerBackend" and "PeerApp". Each entry includes a checkbox, Name (PeerBackend/PeerApp), ARN (arn:aws:elasticloadbalancing:...), Port (80), Protocol (HTTP), Target type (Instance), and a "Details" link. Below the table, it says "0 target groups selected" and "Select a target group above." At the bottom, there are links for CloudShell, Feedback, and footer information including copyright, privacy, terms, and cookie preferences.

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.

The screenshot shows the AWS IAM Roles page. The left sidebar includes 'Identity and Access Management (IAM)', 'Dashboard', 'Access management' (with 'Roles' selected), and other options like 'Policies' and 'Identity providers'. The main content area displays 'Roles (1/22)' with a search bar for 'LabRole' which finds '2 matches'. The first result is 'LabRole' (Account: 776306033623) and the second is 'RoleForLambdaModLabRole' (AWS Service: lambda). Below this, there's a section titled 'Roles Anywhere' with links for 'Access AWS from your non AWS workloads' (X.509 Standard) and 'Temporary credentials'.

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.

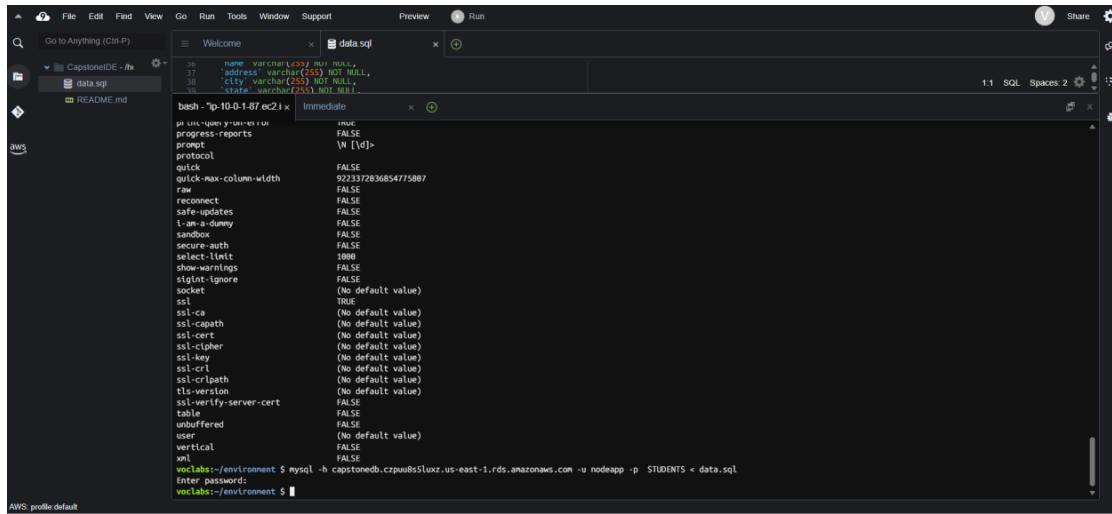
The screenshot shows the AWS Security Groups page. The left sidebar includes 'EC2' (selected), 'Security Groups', and other categories like 'Elastic Block Store', 'Network & Security', 'Load Balancing', and 'Auto Scaling'. A green success message at the top states: 'Security group (sg-012d17449826c426c | ALB_SG) was created successfully'. The main content area displays 'Security Groups (5)' with a table showing columns for 'Name', 'Security group ID', 'Security group name', 'VPC ID', and 'Description'. The groups listed are: default (sg-0206e52fde7873ca8, vpc-0bf16727d18c41c3c), ALB_SG (sg-012d17449826c426c, vpc-073be7fef6ee2d2b2), launch-wizard-1 (sg-002894f63502405e0, vpc-073be7fef6ee2d2b2), default (sg-0ad696a2df556921c, vpc-073be7fef6ee2d2b2), and PeerAppSG (sg-08d9d0625ed2fb76, vpc-073be7fef6ee2d2b2). A 'Select a security group' dropdown is visible at the bottom.

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.



The screenshot shows the AWS Lambda function configuration interface. In the left sidebar, there are tabs for 'CapstoneIDE - /v' and 'data.sql'. The main area displays environment variables and a terminal window.

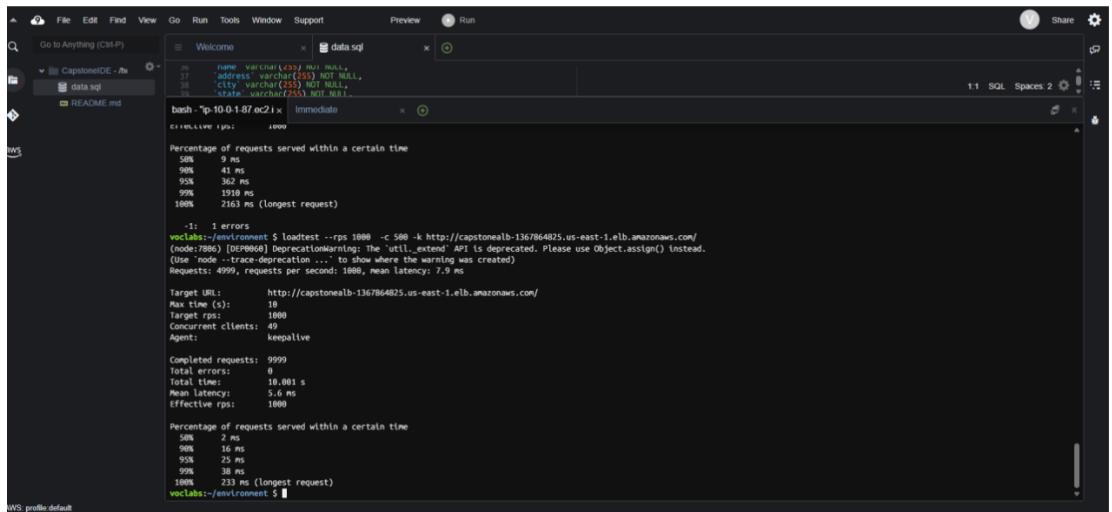
```

{
  "name": "varchar(255) NOT NULL",
  "address": "varchar(255) NOT NULL",
  "city": "varchar(255) NOT NULL",
  "state": "varchar(255) NOT NULL"
}

bash -> ip=10.0.1.87 ec2-i x
  _run_
progress-reports
prompt
protocol
quick
quick-max-column-width
raw
reconnect
safe-updates
lan-a-dubby
sandbox
secure-auth
select-limit
show-warnings
strict-ignore
socket
ssl
ssl-ca
ssl-cipher
ssl-crl
ssl-key
ssl-path
tls-version
ssl-verify-server-cert
table
unbuffered
use
vertical
xml
voclabs:-/environment $ mysql -h capstonedb.czpuu8s5luz.us-east-1.rds.amazonaws.com -u nodeapp -p STUDENTS < data.sql
Enter password:
voclabs:-/environment $ 
  
```

AWS profile default

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



The screenshot shows the AWS Lambda function configuration interface. In the left sidebar, there are tabs for 'CapstoneIDE - /v' and 'data.sql'. The main area displays environment variables and a terminal window.

```

Percentage of requests served within a certain time
50%   9 ms
98%   41 ms
99%   162 ms
100%  1919 ms (longest request)

  1 errors
voclabs:-/environment $ loadtest --rps 1000 -c 500 -k http://capstonealb-1367864825.us-east-1.elb.amazonaws.com/
(node:7886) [DEP0068] DeprecationWarning: The 'util._extend' API is deprecated. Please use Object.assign() instead.
(Use `node --trace-deprecation ...` to show where the warning was created)
Requests: 4999, requests per second: 1000, mean latency: 7.9 ms

Target URL:      http://capstonealb-1367864825.us-east-1.elb.amazonaws.com/
Max time (s):    10
Target rps:      1000
Concurrent clients: 49
Agent:           keepalive

Completed requests: 9999
Total errors:      0
Total time:        10.001 s
Mean latency:     5.6 ms
Effective rps:    1000

Percentage of requests served within a certain time
50%   2 ms
98%   8 ms
99%   25 ms
100%  233 ms (longest request)
  
```

AWS profile default

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

aws pricing calculator

The screenshot shows the AWS Pricing Calculator interface. At the top, there are links for Feedback, Language: English, Contact Sales, and Create an AWS Account. Below the header is a search bar labeled "Find resources". The main content is a table with the following columns: Service Name, Status, Upfront cost, Monthly cost, Description, Region, and Config Sum... (with ellipses). The table lists several services:

Service Name	Status	Upfront cost	Monthly cost	Description	Region	Config Sum...
Amazon Virtual...	-	0.00 USD	73.00 USD	-	US East (N. Virg...)	Working days p...
Amazon EC2	-	0.00 USD	2.92 USD	-	US East (N. Virg...)	Tenancy (Share...
Amazon RDS fo...	-	0.00 USD	95.52 USD	-	US East (N. Virg...)	Storage amoun...
AWS Secrets Ma...	-	0.00 USD	0.40 USD	-	US East (N. Virg...)	Number of secr...
Elastic Load Bal...	-	0.00 USD	22.27 USD	-	US East (N. Virg...)	Number of Appl...

At the bottom of the calculator are links for Privacy, Site terms, Cookie preferences, and a copyright notice: © 2025, Amazon Web Services, Inc. or its affiliates. All rights reserved.

Figure 23: Load Balancer Latency and Request Count Analysis.

The screenshot shows the AWS EC2 Auto Scaling Groups activity history for "CapstoneAutoScalingGroup1". The left sidebar includes navigation links for EC2, Auto Scaling groups, Auto Scaling, and other services like Elastic Block Store and Network & Security. The main area displays an "Activity history [3]" table with columns: Status, Description, Cause, Start time, and End time. The history shows three successful events:

Status	Description	Cause	Start time	End time
Successful	Launching a new EC2 instance: i-0f203c294fc8a94cf	At 2025-11-24T07:35:57Z an instance was launched in response to an unhealthy instance needing to be replaced.	2025 November 24, 12:35:57 PM +05:00	2025 November 24, 12:36:00 PM +05:00
Successful	Terminating EC2 instance: i-04ae0f5dcdb2bb909	At 2025-11-24T07:35:57Z an instance was taken out of service in response to an EC2 health check indicating it has been terminated or stopped.	2025 November 24, 12:35:57 PM +05:00	2025 November 24, 12:41:11 PM +05:00
Successful	Launching a new EC2 instance: i-04ae0f5dcdb2bb909	At 2025-11-24T06:02:23Z a user request created an AutoScalingGroup changing the desired capacity from 0 to 1. At 2025-11-24T06:02:24Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 0 to 1.	2025 November 24, 11:02:25 AM +05:00	2025 November 24, 11:02:35 AM +05:00

Figure 24: Successful Response Validation under Peak Load Conditions.

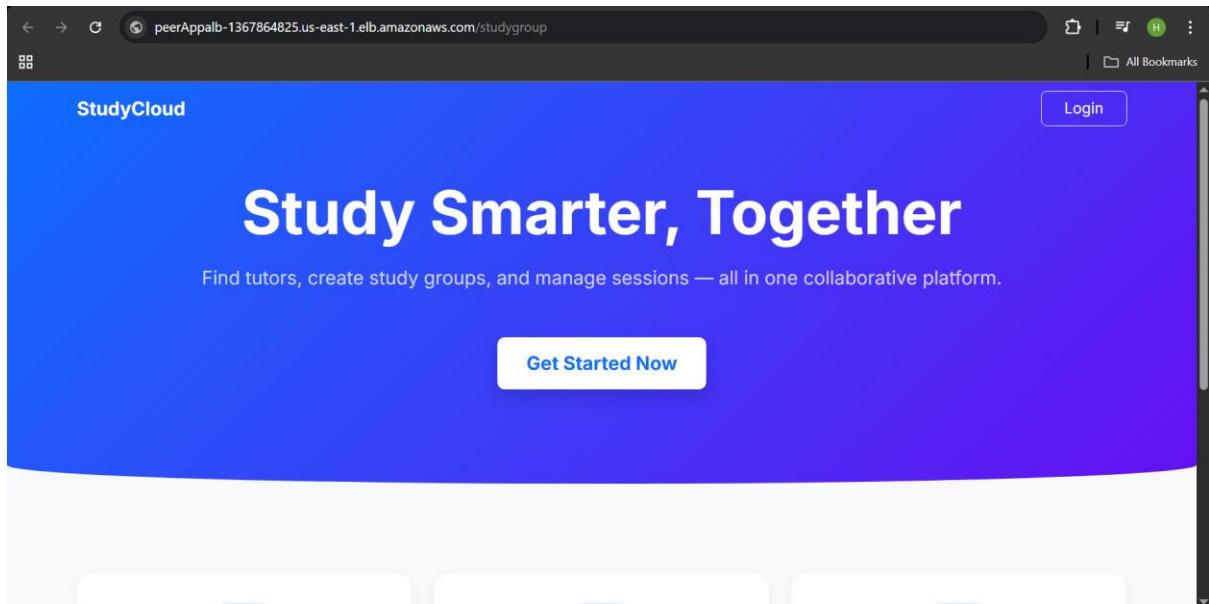


Figure 25 : Final System Health Check and Blue-Green Deployment Completion with ALB

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.