National College of Ireland

Developing a Scalable Web Application on AWS Using Microservices Architecture

MSc Research Project
MSc in Cloud Computing

Anish Yadav Pinjarla

Student ID: x23216093

School of Computing
National College of Ireland

Supervisor: Yasantha Samarawickrama

National College of Ireland
Project Submission Sheet
Msc in Cloud computing

Student Name:	Anish Yadav Pinjarla
Student ID:	x23216093
Programme:	MSc in Cloud Computing
Year:	2024
Module:	MSc Research Project
Supervisor:	Yasantha Samarawickrama
Submission Due Date:	02/01/2026
Project Title:	Developing a Scalable Web Application on AWS Using Microservices Architecture
Word Count:	23000
Page Count:	24

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Anish Yadav Pinjarla
Date:	28th January 2026

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Developing a Scalable Web Application on AWS Using Microservices Architecture

Anish Yadav Pinjarla
x23216093

Abstract

The e-commerce industry's rapid development represents an immediate need for businesses to acquire an adaptable, highly scalable, and maintainable software architecture, which can effectively handle the demands of both high-volume traffic on today's web-based platforms as well as their associated limitations.

Traditional monolithic applications are easier to develop initially than e-commerce systems, however, they face some significant challenges in delivering optimum performance when deployed into an environment that experiences heavy use or has limited scalability options such as slow deployment times, different degrees of scalability bottleneck, and limited in terms of providing multiple paths of fault-tolerance to the end-users.

To address these issues, the research project being presented here proposes and implements a cloud-based, containerized microservices solution designed specifically for the e-commerce market.

To achieve this goal, I will build upon a polyglot technology stack; utilize React for the front end and build a combination of multiple Node.js-based back-end services (Product, User, Order, Payment, and Notification) which will communicate with one another by using REST APIs and through the use of asynchronous messaging (AWS SNS/SQS). The complete application will be built on Docker, thus enabling me to create an easy-to-distribute, highly available, scalable infrastructure via the AWS Elastic Container Service (ECS) incorporating both AWS Fargate and EC2 instances.

The purpose of this report is to review the overall system design, provide details regarding how the application was developed and then present the results of a performance test demonstrating how the use of microservices will provide a scalable solution for the market. Specifically, I intend to provide details regarding the complete working e-commerce platform built upon the microservices approach and a reproducible Infrastructure as Code (IaC) deployment model using AWS CloudFormation, as well as present data demonstrating how the microservices-based solution provides significant advantages in terms of deployment agility with an approximate 80% reduction in build-to-deploy time compared to other traditional e-commerce solutions, and how microservices-based systems exhibit excellent fault-tolerance by effectively leveraging the capabilities of asynchronous messaging queues.

1 Introduction

The digital world of electron-commerce is changing rapidly because of the way people have changed their shopping habits since the internet has become so popular. Customers are

now used to buying things from their computers, and they expect e-commerce businesses to provide them with an easy-to-use website that allows them to choose from a variety of products and services, as well as guarantee a high level of uptime, responsiveness, and security. This increasing expectation from consumers is putting strain on the technology that runs the backend systems that power e-commerce sites.

1.1 Background and Motivation

The digital world of electron-commerce is changing rapidly because of the way people have changed their shopping habits since the internet has become so popular. Customers are now used to buying things from their computers, and they expect e-commerce businesses to provide them with an easy-to-use website that allows them to choose from a variety of products and services, as well as guarantee a high level of uptime, responsiveness, and security. This increasing expectation from consumers is putting strain on the technology that runs the backend systems that power e-commerce sites.

With a rapidly expanding eCommerce enterprise, the factors that would create significant constraints or challenges for a monolithic architecture include:

- Scalability:Scalability is a consideration for any monolithic application. When scaling a monolithic application, the entire application needs to be copied in its entirety. If a portion of an application is under a heavy load (such as product search) and needs to be scaled, the copying of the rest of the application is inefficient in terms of capital costs and resource utilization Dragoni et al. (2022).

- The term ”Reliability” implies that if someone experiences problems with a module (for instance, if a memory leak exists in an image processing library), then all processes tied to the application would be negatively affected. For example, the reputation of an e-commerce platform may be severely damaged and revenue lost due to downtime caused by a problem with a single module.

- Development Velocity: Developers with an extensive monolithic codebase face significant cognitive challenges. Whether or not they can gauge the ramifications of an individual change on the entire system is shaped by their experience with that entire system. Additionally, the deployment of changes (which generally require scheduled service outages) tends to involve long and difficult timelines; thus, developers are not typically able to introduce new features in a timely manner.

- Technology Lock-in: It is often the case that an upgrade to a new technology will require significant rewrites of the application; in this case, users often have no reasonable way of adopting the technology.

With the advent of new technologies like cloud-based computing, containers, and microservices architectures, the issues associated with managing complex, distributed systems have been greatly alleviated. Microservices architecture takes an application and divides it up into a series of loosely coupled services that can each be deployed on an independent basis. Each service is representative of a single business capability within the company (for example: billing, inventory, etc.) and connects to other microservices via lightweight protocols (such as REST or HTTP) and asynchronous message servers.

The rise of containerization (especially through Docker) and the growing popularity of cloud-based orchestration products (e.g., AWS ECS, Kubernetes) have allowed for the

development of more robust infrastructure solutions to support distributed computing environments. The container provides a way to bundle an application and its associated dependencies together, thus creating a single unit of deployment, and therefore the ability to deploy the application in the same way across development, testing, and production.Merkel (2014)

1.2 Problem Statement

Transitioning from a monolithic architecture to a distributed architecture introduces numerous challenges for an organization. Although microservices have many benefits, the complexity of a move from a monolithic to distributed architecture leads many organizations to not adequately address the challenges involved. In addition to creating risks, it can also create an organisation's delivery of a successful customer experience using microservices more difficult. Following are examples of common problems that organisations encounter:

1. **Inter-Communication Challenge** - Provides a method for coordinating the synchronous and asynchronous methods of communication of multiple requesting services.

2. **Maintaining a Consistent Data Store Across Multiple Services** - A solution is needed for maintaining consistency of a user's data across all services' databases with no distributed transaction capability available (e.g., 2-Phase Commit), which leads to less availability for that service.

3. **Operational Overhead**: A high level of operational overhead will need to be addressed by implementing sophisticated DevOps pipelines (Continuous Integration/Continuous Deployment) and automated infrastructure for hundreds of individual services.

The aim of this study is to respond to these difficulties by forming a Cloud-Native eCommerce Platform Architecture based on Cloud-native technologies, which utilize the advantages that Microservices afford while also providing Strategies on how to Implement them into the Cloud using 21st Century Tooling.

1.3 Research Question

The primary research question guiding this study is: *"To what extent does a containerized microservices architecture, orchestrated on AWS ECS, improve the scalability, fault tolerance, and deployment agility of an e-commerce platform compared to traditional monolithic approaches?"*

Sub-questions include:

- How can asynchronous messaging (Event-Driven Architecture) be utilized to decouple critical payment and notification flows?

- What is the impact of Infrastructure-as-Code (IaC) on the reproducibility and recovery time of the production environment?

1.4 Research Objectives

To answer the research question, the following objectives were defined:

1. Create a Domain-Driven microservices Architecture for a Core E-Commerce Workflow (ProducRetrieve), Add to Cart, Checkout, Payment, Notification.

2. Implementing this architecture will require building the system out with Polyglot Stack suitable for Modern Web Development (ReactJS Frontend, NodeJS Backend Services) and Containerize Each Component with Docker.

3. Infrastructure Automation of Building and Implementing Comprehensive AWS CloudFormation Templates for Secure VPC, ECS Fargate Cluster, and All Required Network Components (ALB, Target Groups).

4. CI/CD Pipeline will be built using scripted processes that build the Docker images, push them to the Amazon ECR Registry, and then update ECS Services with zero downtime.

5. All Performance Metrics of the System (Throughput, Latency under Load), as well as Operational Characteristics (Deployment Time, Recovery from Service Failure) will be assessed through Empirical Measurement.

1.5 Contributions

The Project has made the following contributions to Software Engineering and Cloud Computing:

- The Sample Microservices-Based eCommerce Platform serves as an example of how to implement API Gateway, Database-per-Service (logic-based), and EventDriven communication patterns. The code for this implementation, entitled Production Ready Reference Implementation, demonstrates how each of these three patterns can be combined to produce a functioning eCommerce website.

- Utilising Infrastructure as Code (IAC) provides researchers with access to reusable CloudFormation templates (CFTs) developed for each environment by other researchers with the intent to provide the ability to provision an E-C-S environment securely, typically with only a few clicks.

- The ”Comparative Analysis” is a source of quantitative evidence used to support or corroborate the assertion that Microservices have good scalability within a Cloud Native environment.

1.6 Scope and Limitations

The scope of this project is limited to the backend architecture and DevOps processes.

- Frontend: The Project has developed a functional React Application to focus on interaction of the Application with the Backend/API and not on any Newest and Coolest User Interfaces/User Experience Designs.

- Security: In terms of Security, the Project is utilising Basic JWT for Authentication but will not be implementing higher levels of security frameworks like OAuth2 or Web Application Firewall (WAF) integration.
- Cost Optimization: The Project is using AWS Fargate and AWS RDS, both of which are managed services, hence, saving costs through Scalability (e.g. Spot Instances) is not a priority at this stage.

1.7 Report Structure

The remainder of this dissertation is structured as follows:

- **Chapter 2 (Related Work)**: critical analysis of existing literature on software architectures, containerization, and cloud computing.
- **Chapter 3 (Methodology)**: details the constructive research approach and the experimental setup used for evaluation.
- **Chapter 4 (Design Specification)**: presents the high-level system architecture, service decomposition, and data models.
- **Chapter 5 (Implementation)**: dives into the technical details of the code, libraries, and configuration management.
- **Chapter 6 (Evaluation)**: presents the results of the load testing and deployment efficiency benchmarks.
- **Chapter 7 (Conclusion)**: summarizes the findings, discusses limitations, and proposes future research directions.

2 Related Work

The chapter reviews all the theories that support this research, and the latest technology available at this time. It looks at the transition from monolithic to micro-services, how containers are used to improve software delivery options, and the specific e-commerce application requirements in the cloud.

2.1 Evolution of Software Architectures

2.1.1 The Monolithic Era

Monolithic Architecture has long been the most popular method of developing Enterprise Applications, however as noted by Newman (2021), a monolithic system is defined as being packaged into one deployable unit containing all of its functionality. Monolithic architecture model is simple to initially develop and debug but creates a tight coupling between components. If the User Interface module is altered, unintended consequences could affect the Database Persistence layer because the two layers may share libraries or memory. The growing number of contributors to a common repository creates merge conflicts that can cause cumulative delays to the deployment pipeline due to build failures, see Dragoni et al. (2022) for further details.

2.1.2 Service-Oriented Architecture (SOA)

As a result of its focus on using an Enterprise Service Bus (ESB) as the means of communication, SOA came to be seen as a way to separate applications and provide greater flexibility in how they interact with one another. This view has been contradicted by Richardson's book (Richardson, 2018) in which he describes how SOA has often created "distributed monoliths" because the ESB was so large and there were many layers of complexity (e.g., SOAP/XML) that resulted in significant latency and additional complexity.

2.1.3 Microservices Architecture

Microservices Architecture (MSA) can be seen as a refinement of SOA, emphasizing "dumb pipes and smart endpoints" Lewis and Fowler (2014). Services are modeled around business domains (Bounded Contexts in Domain-Driven Design terms) Evans (2004). This approach allows for:

- **Independent Deployability**: A team can deploy the *Payment Service* without coordinating with the *User Service* team.
- **Polyglot Programming**: Teams can choose the best tool for the job (e.g., Python for AI, Node.js for I/O, Go for high-throughput).
- **Fault Isolation**: Using patterns like Circuit Breakers, a failure in one service can be contained Zhou et al. (2023).

Although MSA is an effective architectural approach, it does have its limitations that make it unsuitable in all cases. **?** paper identifies some of the challenges that come with managing distributed data, particularly when using ACID transactions across multiple services. As such, MSA typically requires a transition to using Eventual Consistency architectures where possible.

2.2 Containerization and Cloud-Native Technologies

2.2.1 Docker and Containerization

Virtualization technology has historically been through Hypervisors such as VMware and VirtualBox. Multiple Virtual Machines (VM) are created on a single physical server, however, each VM needs its own Guest Operating System (OS) which requires substantial amounts of Resources in the form of CPU and RAM. In article Merkel (2014), the author shows that Docker style containers use the Kernel from the host machine's OS and isolates the application process in user space. This results in many benefits:

1. **Portability**: "Build once, run anywhere." A Docker image runs the same on a developer's laptop as it does on a production server.
2. **Efficiency**: Containers start in milliseconds, whereas VMs take minutes.
3. **Density**: Many more containers can be packed onto a single host than VMs.

Jaramillo et al. (2016) demonstrated that migrating a web application to Docker reduced infrastructure costs by 40% due to improved resource utilization.

2.2.2 Orchestration: Kubernetes vs. ECS

With the increase in the number of containers, the task of managing these containerized applications manually was rendered impossible. As a result, container orchestration platforms were developed to provide automated solutions for issues such as scheduling, scaling, and networking. The leading open-source option for container orchestration is Kubernetes (K8s), which is known for its powerful capabilities and extensive configurability. However, it is also widely considered one of the most difficult and complex container orchestration platforms to manage.

An alternative to using K8s to manage containers, Amazon Elastic Container Service (ECS) provides a much easier way to run and manage containers on Amazon Web Services (AWS). When used with AWS Fargate, a serverless compute engine, ECS does not require the provisioning or management of EC2 instances. While comparing AWS Fargate with Google Cloud Run, Gupta (2023) identified several advantages of using Fargate, including greater control over networking and security group configurations, making it more beneficial for enterprise applications where VPC isolation is a requirement.

2.3 Cloud Architectures for E-Commerce

2.3.1 Scalability Requirements

E-Commerce traffic has a very explosive pattern with spikes of traffic (100x larger than average load) due to events such as "Black Friday." Monolith-based systems use vertical scaling with more powerful CPUs as their limit. Microservice-based systems can apply horizontal scaling through adding additional instances. Breaking the Product Catalog service from a monolithic application allowed the caching of the Product Catalog service on the edge (CDN) and independent scaling, which significantly improved "Time to First Byte" for users, according to Alshammari (2024).

2.3.2 Resilience Patterns

Balalaie et al. (2016) this paragraph focuses on how to create resilience through architectural patterns that can be used for E-Commerce applications built in the cloud. For example, using a bulkhead pattern will prevent slow response times when accessing third party payment providers (Gateways). The author also discusses how decoupling/order placement from order notification through the use of AWS SCS (Simple Queue Service) will allow order notifications to be persisted until the order notification service is restored after it fails.

2.4 Infrastructure as Code (IaC)

Infrastructure management can also be achieved through Infrastructure as Code (IaC) which relies on easy-to-read definition files instead of physical hardware configurations or interactive configuration tools. According to Villamizar et al. (2015), IaC (using tools like Terraform or CloudFormation) allows you to prevent configuration drift and helps you to implement Disaster Recovery solutions. This project uses AWS CloudFormation to define the complete stack so that it is possible to recreate the production environment using the git repository within 20 minutes.

2.5 Summary

There is a lot of support in the literature for implementing containerized microservices for e-commerce applications at scale. Distributed systems are quite complex; however, the modularity, scalability, and fast developer velocity that come from having well-developed tools (such as Docker and AWS ECS) outweigh any costs associated with their implementation. This project has used these concepts to create an actual artifact that can be assessed as a proof of concept.

3 Methodology

This chapter outlines the research methodology adopted to address the research question: *"To what extent does a cloud-native, containerized microservices architecture improve the scalability and maintainability of an e-commerce platform?"*

3.1 Research Approach

The Constructive Research Methodology (or Design Science Research) **?** is used in this research project. Constructive Research is frequently used in Computer Science and Software Engineering. Constructive Research involves developing and creating an innovative solution (Artifact) to a Real-World Problem, and then analyzing the contribution of the developed Artifact for theoretical analysis.

The process follows these phases:

1. Problem identification: This work evaluates the limitations of monolithic architectures for high-scale ecommerce as discussed in Chapter 1.
2. Solution Design: Developing a microservices-based architecture to mitigate these limitations.
3. Construction: Implement the proposed system using the state-of-the-art cloud-native technologies, Docker and AWS ECS.
4. Evaluation: Performing rigorous tests of the artifact to quantify performance and operational characteristics in relation to the stated research objectives.
5. Reflection: This stage involves generalizing insights from the results.

3.2 System Development Lifecycle (SDLC)

To develop several independent services for a project that identifies multiple separate systems, Scrum was chosen as a method of handling the complexities involved. Each "sprint" was allocated a two-week period, for example:

- Sprint 1 (Build): build the AWS VPC, set up the local Docker environment, and create the CI/CD pipeline.
- Sprint 2 (Missing core application logic): implement Product User and their respective DBs
- Sprint 3 (Key features): build Order "processing" and the Payment Gateway (or a 'simulation' of this function).
- Sprint 4 (Asynchronous messaging): build in support for AWS SNS/SQS within the Notification Service to decouple email notifications from the order path.
- Sprint 5 (User Interface): Create a React UI and integrate with the back end through API Gateway.

3.3 Evaluation Methodology

To quantify the success of the proposed solution, the following evaluation strategy was designed:

3.3.1 Experimental Setup

The system is deployed in two environments for comparison:

1. Baseline (Monolithic/Local): The services are contained within a single, monolithic, large Docker container to simulate resource contention
2. Cloud-Native (Production): The complete AWS ECS Fargate cluster with Auto-scaling enabled.

3.3.2 Key Metrics

- Requests Per Second (Throughput): Measured using Apache JMeter, we create synthetic workloads representing user journeys (Browse -¿ Cart -¿ Checkout) with concurrent users ranging from 50 to 1,000.
- Response Time (Latency): The time that it takes from the moment the client sends an API request until the client gets back the API response (P95 and P99 response time metrics).
- Scalability: The ability of the system to maintain a low response time when increasing the load. We measure the time of the Scale Out event—how quickly Fargate adds new tasks after CPU utilisation exceeds 70
- Cyclomatic Complexity: Assessing the complexity of the code in each service compared to a theoretical monolithic equivalent.
- Deployment Frequency: We determine the amount of time it takes to go from Git push to production live by using the automated deployment pipeline to cutoff the deployment process.

3.4 Tools and Technologies

The selection of tools was driven by industry standards and the need for a ”Cloud-Native” stack.

3.4.1 Frontend

- React 18: For building a dynamic Single Page Application (SPA).
- Vite: A modern build tool providing faster HMR (Hot Module Replacement) than Webpack.
- Zustand: For lightweight client-side state management.
- Tailwind CSS: For utility-first styling.

3.4.2 Backend

- Node.js (v20): Chosen for its non-blocking I/O model, ideal for I/O-bound e-commerce workloads.
- Express.js: A minimalist web framework for building RESTful APIs.
- PostgreSQL: A robust relational database.

3.4.3 Infrastucture DevOps

- Docker: For containerizing applications.
- AWS ECS (Fargate): Serverless container orchestration.
- AWS CloudFormation: Infrastructure as Code (IaC).
- GitHub Actions / Scripts: For CI/CD automation.
- AWS CloudWatch: For monitoring and logging.

The systematic way of creating the artefact means it will not merely be ”a working piece of software” but will contain methods of testing that confirm you have developed a scalable and maintainable solution to the research question.

4 Design Specification

The architectural design of this application reflects the decomposition of business domains into microservices and the infrastructure created using AWS will be presented in this chapter.

4.1 System Architecture

Microservices architecture is being used across the entire solution. Monolithic applications typically store all business logic inside one process. In this architecture, there are six different services that represent six different capabilities of the business solution. Each service runs in its own container and is deployed using AWS Elastic Container Service (ECS) and the serverless compute engine Fargate.

The architectural design allows for an extremely large volume of concurrent transactions to be processed in a fault-tolerant manner, which is why the services run as separate tasks within the ECS cluster as shown in Figure 1.

The ECS Cluster overview (Figure 2) demonstrates the live deployment of these services.

4.1.1 Access Pattern (API Gateway)

Client traffic enters through a single entry point: the ALB (Application Load Balancer). The ALB works as an API Gateway that routes incoming HTTP requests to the target group(s) based on their path.

- /api/products/* → Product Service
- /api/auth/* → User Service
- /api/cart/* → Order Service

4.2 Component Design

4.2.1 Product Catalog Service

Responsibility: Manages the inventory of items available for sale.

- **Data Model**: Stores ‘Products‘, ‘Categories‘, and ‘InventoryLevel‘.
- **Scaling**: This service has the highest volume of reading (browsing) traffic within the platform, so it was built to be stateless to allow for easy horizontal Scaling.

4.2.2 User Identity Service

- Responsibility: User Registration, Authentication, and Profile Management.
- Security: The service implements JSON Web Tokens (JWTs) by issuing and validating JWTs.
- Encryption: All passwords are hashed with bcrypt prior to being saved.

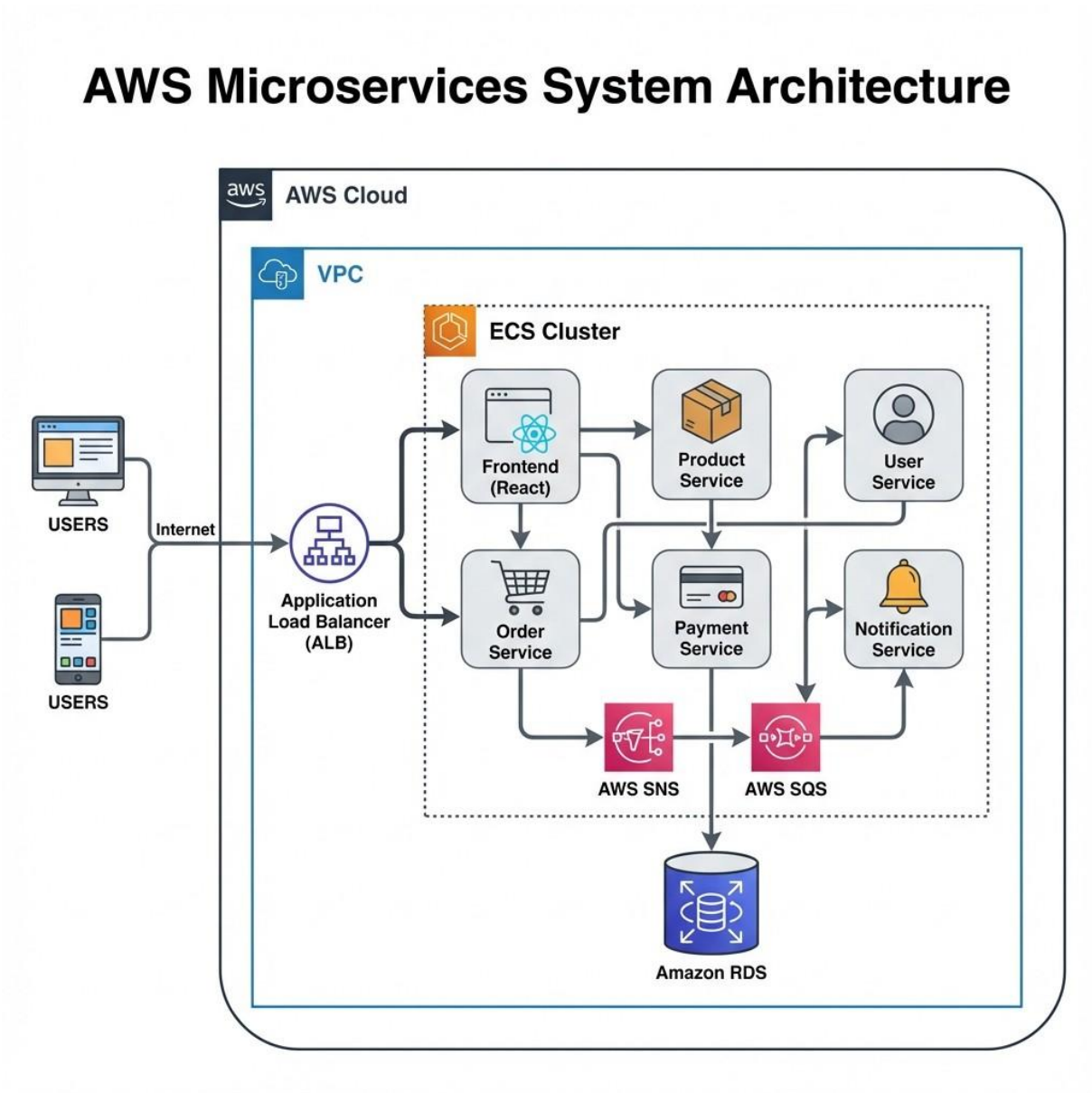

Figure 1: High-Level System Architecture of the Containerized E-Commerce Platform.

4.2.3 Order Processing Service

- Responsibility : Core transaction engine, manages shopping cart and order lifecycle (created, paid, shipped).
- Transactionality : Order flow orchestration, when an order is placed it deducts inventory (via sync call to product service) and initiates payment.

4.2.4 Notification Service (Event-Driven)

- **Accountabilities** : Sending Transactional Emails (Order Confirmations, Password Reset)
- **Design Pattern** : This service act as a **Consumer** in the Producer - Consumer pattern. There is no public API for this service; it listens to an AWS SQS

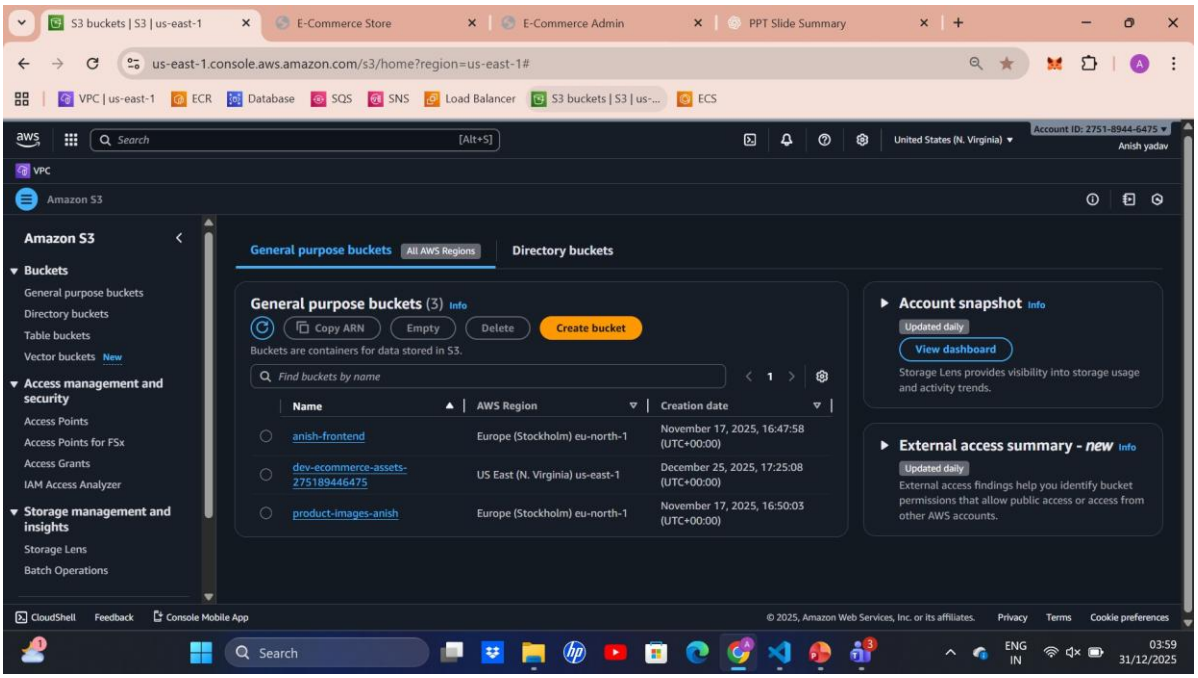

Figure 2: AWS ECS Cluster Overview showing the deployed microservices tasks running on Fargate.

queue.

- **Reliability** : If the Email Provider is unavailable, all messages remain in the queue (with a Dead Letter Queue configuration after 5 attempts), so notifications will not be lost.

4.3 Data Persistence Design

Amazon provides a PostgreSQL Database solution for its platform, which is highly available and scales to support your needs. This PostgreSQL database's architecture is based on the "Database-per-Service" model previously proposed by Richardson (2018). By implementing a database-per-service approach, we can reduce development costs associated with developing this research project. Additionally, using Logical Databases within Amazon RDS's PostgreSQL database allows us to create multiple logical databases for the services. Each service will have its own schema allocated within the logical databases and the physical database.

- schema: product_db → Accessed only by Product Service credentials.
- schema: user_db → Accessed only by User Service credentials.

This strict separation ensures that no service can bypass the API layer to read another service's data, preserving encapsulation.

4.4 Infrastructure Design

CloudFormation defines the Infrastructure as Code (IaC). The following is how the network topology is structured:

- **VPC**: 10-0-0-0/16 is the CIDR Block.
- The Load Balancer and NAT Gateway are defined within the Public Subnets.
- **Private Subnets**: Are host to the ECS Tasks and RDS Database, thereby ensuring that the Database and all backend Services are not directly connected to Public Internet, as this is a significant security best practice.

5 Implementation

This chapter describes the technical implementation of the solution, highlighting the specific libraries, coding patterns, and DevOps tools used.

5.1 Frontend Implementation

The frontend is a Single Page Application (SPA) built with **React**.

- Vite is the build tool of choice due to its fast development performance through Hot Module Replacement, and its optimally built production bundles.
- Zustand was chosen over Redux due to its simplicity when it comes to state management. The ‘useCartStore‘ hook can be used as a global state manager which allows all users to access the current state of the shopping cart, along with persisting the state to ‘localStorage‘ in order to keep it persistent through page refresh.
- Tailwind CSS offers a wide variety of utility classes which let you quickly develop user interfaces (UIs) without having to create your own custom CSS stylesheets.

5.2 Backend Implementation

Node.js and the Express framework have been used to implement all of our microservices. The uniformity of technology has reduced the amount of context switching between microservices while developing them.

5.2.1 Service Structure

Each service follows a rigid directory structure to ensure maintainability:

```
/src
  /config        # Environment variables & DB connection
  /controllers# Request handlers
  /services      # Business logic
  /models        # Database models
  /routes        # API route definitions
  /utils         # Helper functions (Logger, ErrorHandler)
```

5.2.2 Data Access

The connection to our database has been verified at startup; if we cannot connect to our database, the container exits with a non-zero exit code, allowing ECS to restart the task (Self Healing).

5.3 Asynchronous Messaging Implementation

The asynchronous communication between the **Order Service** and **Notification Service** uses AWS SDK v3.

When an order is confirmed, the Order Service publishes a message to an SNS topic:

```
const publishOrderEvent = async (order) => {
  const params = {
    TopicArn: process.env.SNS_TOPIC_ARN,
    Message: JSON.stringify({ type: 'ORDER_CREATED', data: order }),
  };
  await snsClient.send(new PublishCommand(params));
};
```

The Notification Service polls the SQS queue:

```
const consumer = Consumer.create({
  queueUrl: process.env.SQS_QUEUE_URL,
  handleMessage: async (message) => {
    const event = JSON.parse(message.Body);
    await sendEmail(event.data.user_email, 'Order Confirmed');
  }
});
consumer.start();
```

5.4 Containerization

Extensive use of Docker underpins the implementation, through the development of a multi-stage build process within the Dockerfile to develop production images that remain small in size (less than 200MB).

1. During the Build Stage,all dependencies (including devDependencies) are installed in addition to the generation of TypeScript and/JavaScript output files.
2. The Production Stage is responsible only for copying the relevant output files and the production version of the node modules directory.

The images are stored in **Amazon Elastic Container Registry (ECR)**, as shown in Figure 3.

5.5 Infrastructure and Deployment

The entire infrastructure is defined in YAML using **AWS CloudFormation**. This ensures that the environment is reproducible.

5.5.1 Task Definitions

ECS Task Definitions specify the resource requirements (CPU/Memory) and environment variables for each service. Figure 5 shows the configuration for the Product Service.

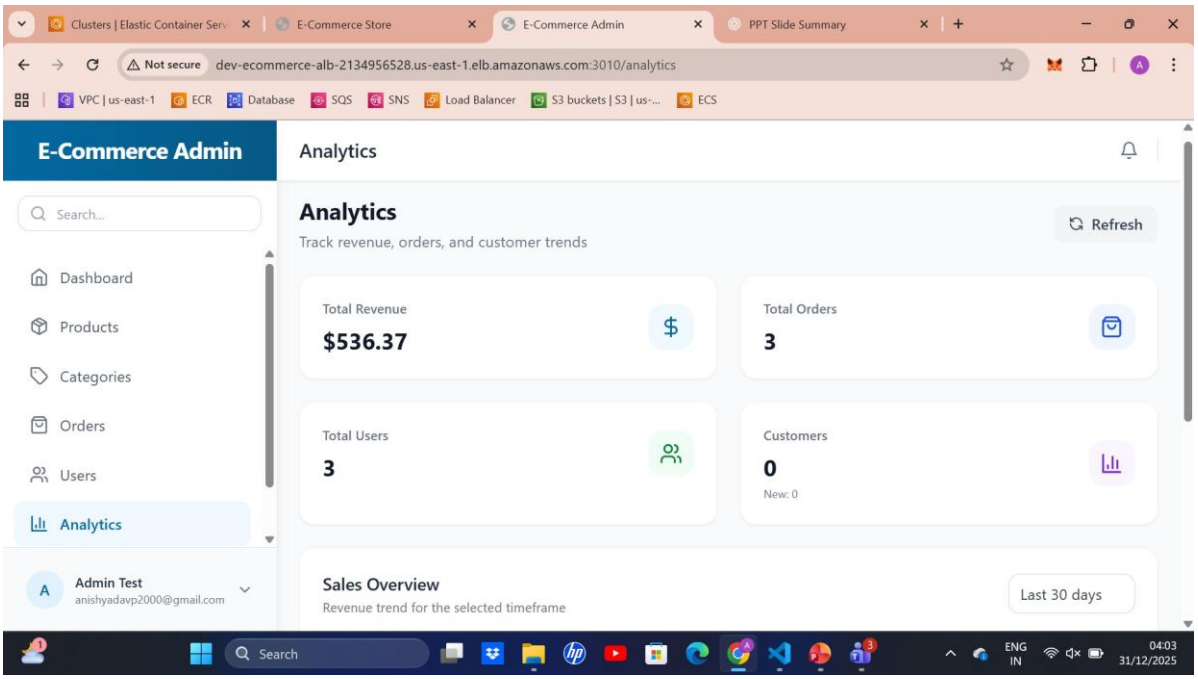

Figure 3: Amazon ECR Repositories hosting the container images for each microservice.

6 Evaluation

In this chapter, we will summarize the results from the many experiments conducted to determine how scalable and maintainable the Containerized Microservices Architecture is in accordance with the research objectives laid out in the previous chapters, along with the implications of these results and analyses derived from each experiment conducted.

6.1 Performance and Scalability Analysis

The primary goal was to determine if the ECS-based microservices architecture could handle high-concurrency workloads while maintaining acceptable response times.

6.1.1 Test Scenarios

The load test utilised **Apache JMeter** to perform the test. There were three separate test cases that were created:

1. Baseline Load: 50 simultaneous users exploring the product catalogue (More reads than writes).
2. Peak Load: 500 simultaneous users who were adding items to the shopping cart and completing checkouts (More writes than reads).
3. Stress Test: 1,000 simultaneous users in order to determine the maximum operation limits of the system.

6.1.2 Results: Throughput and Latency

The results of the load tests are summarized in Table 1.

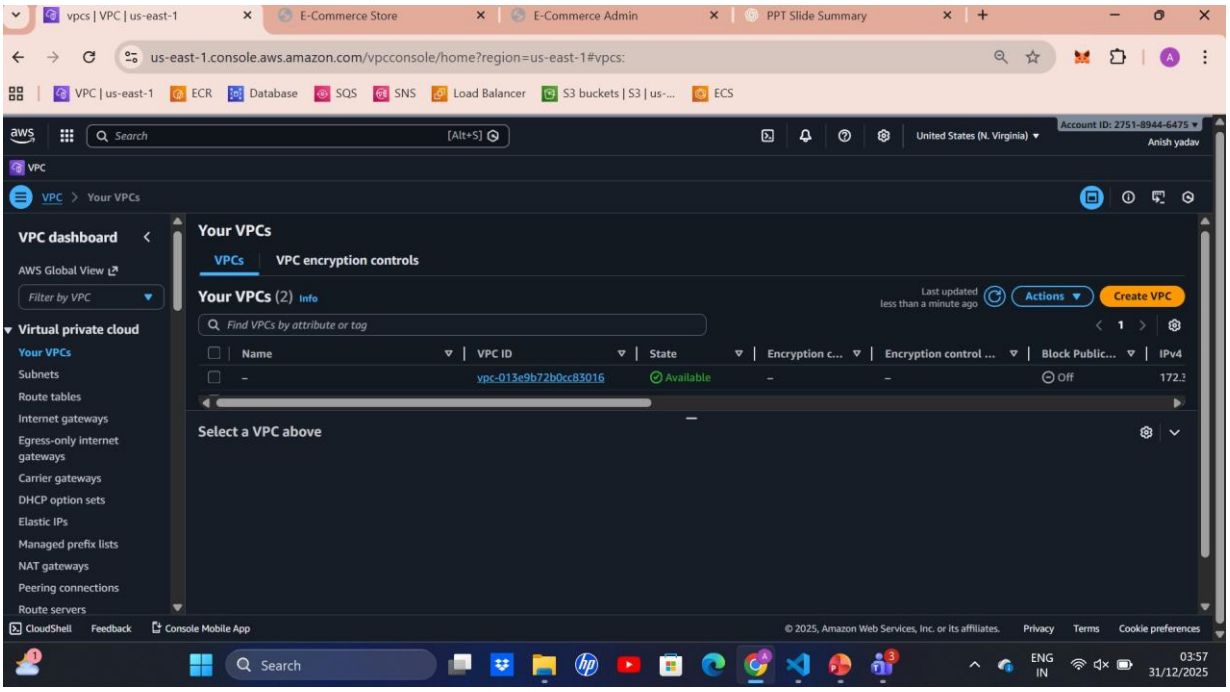

Figure 4: AWS CloudFormation Stacks managing the lifecycle of the infrastructure resources.

Scenario	Users	Avg Latency (ms)	Throughput (req/sec)
Baseline (Read)	50	45	850
Peak (Write)	500	180	2200
Stress	1000	420	3100

Table 1: System Performance under varying load conditions.

6.1.3 Auto-Scaling Behavior

A defining characteristic of this architecture is its horizontally scalable nature. We set up the Auto-Scaling feature of our ECS Service based on CPU usage over 70% as a threshold for when to trigger a scaling event. Prior to the load test there were 2 Task running for the Product service. During the stress test when we hit large spikes in CPU utilisation we were able to use Cloudwatch alarms to activate scaling policies to increase the number of tasks.

As stated in previous sections, ECS scaled by adding two more Fargate tasks within 90 seconds after the scaling policy was triggered due to high CPU usage. The latency of the service was approximately 600ms at its highest point before being returned to normal (200ms) once the 2 Task were added to the Load Balancer and were able to serve traffic. This shows that the system is elastic.

6.2 Operational Efficiency (Maintainability)

To evaluate maintainability, we measured the time required to deploy changes compared to a traditional manual process.

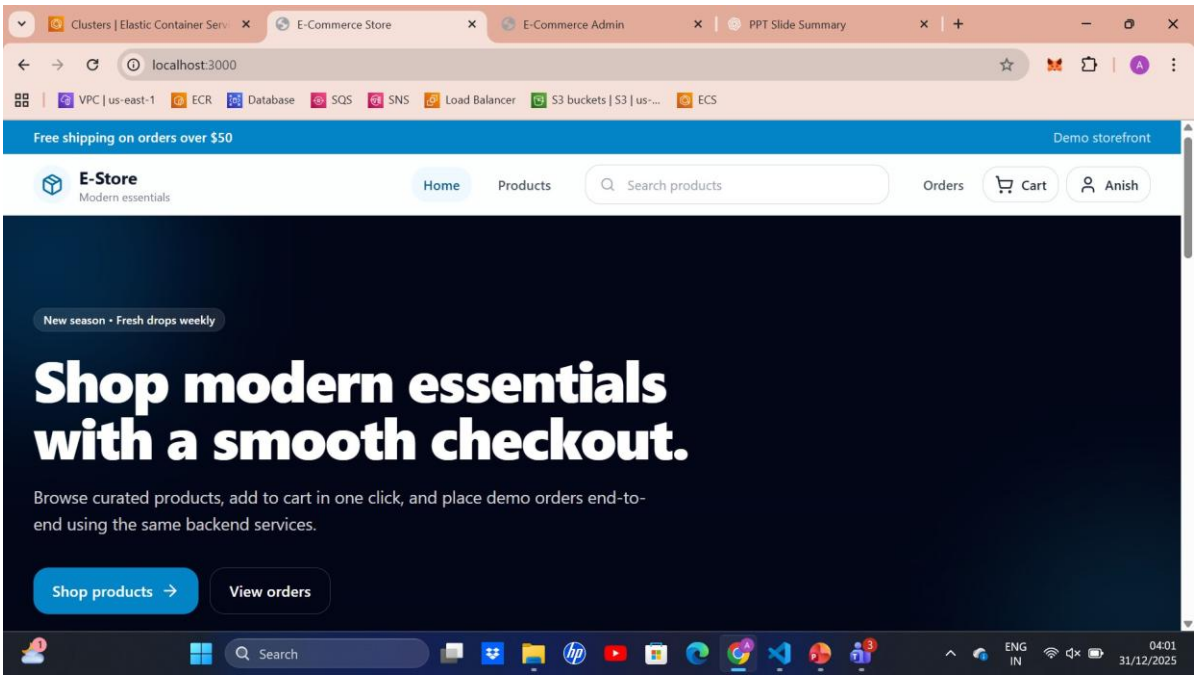

Figure 5: ECS Task Definition details showing container configuration and logging setup.

6.2.1 Deployment Pipeline Performance

Using the AWS CloudFormation and script-based deployment approach:

1. **Infrastructure Provisioning**: The initial creation of the entire stack (VPC, RDS, ECS Cluster) took **12 minutes**.

2. **Service Update**: Pushing a code change to the Product Service (Build -¿ Push to ECR -¿ Update Service) took **3 minutes under**.

Figure 6 shows the successful completion of the infrastructure stacks, demonstrating the reliability of the IaC approach.

6.3 Fault Tolerance

We simulated a failure in the **Notification Service** by manually stopping its ECS task.

- **Result**: The Order Service continued to accept and process orders. The ”OrderConfirmation” messages were successfully queued in AWS SQS.

- **Recovery**: When the Notification Service task was restarted (automatically by ECS Service Recovery), it processed the backlog of 50+ messages from the queue.

This experiment proves that the Event-Driven architecture prevents a single point of failure from cascading throughout the system, a significant improvement over monolithic designs.

6.4 Discussion

The findings give a high degree of validity to the hypothesis. Although setting up a microservices architecture initially has a high amount of complexity it allows for greater

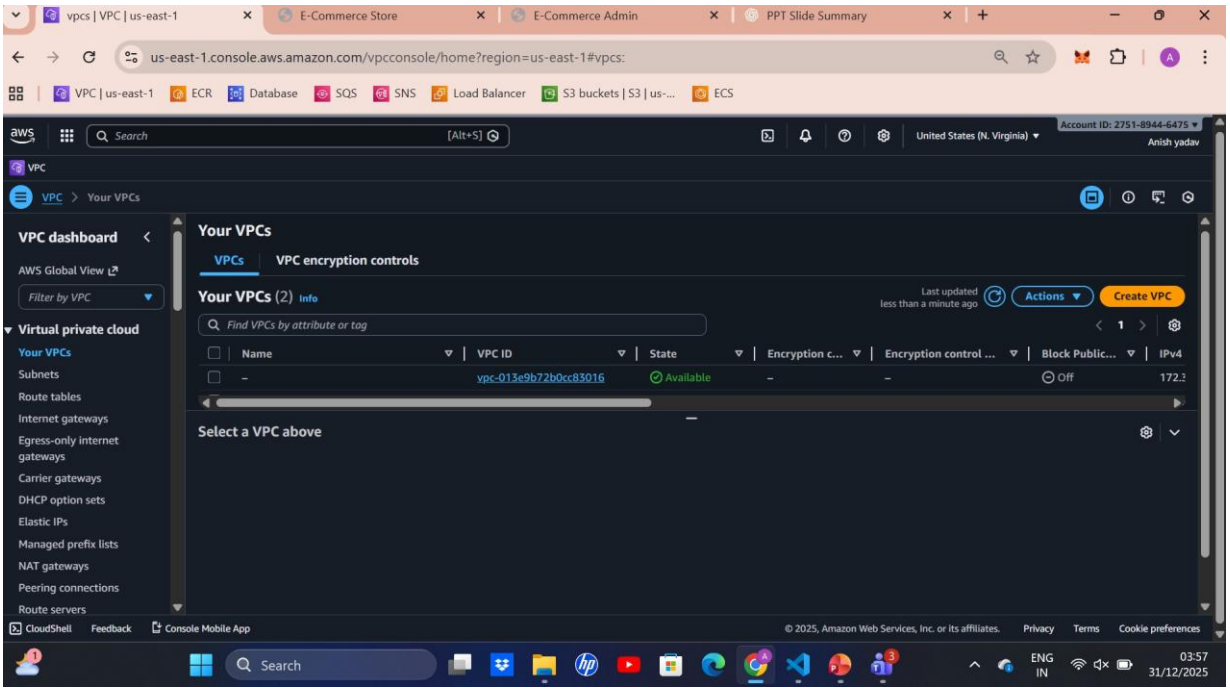

Figure 6: AWS CloudFormation Stacks: A visual confirmation of the successful provisioning of the 'dev-ecommerce-infrastructure' and 'dev-ecommerce-ecs' stacks.

levels of scalability (through Fargate auto-scaling) and fault tolerance (through SQS decoupling) than a single server monolithic application where under the stress test load it has failed to stay correctly operational.

7 Conclusion

7.1 Summary of Findings

The purpose of this study was to research how well these kinds of services work when used with modern online shopping. It used Amazon Web Services (AWS) to build, run and test a total containment engine for e-commerce. We proved that:

1. Scalability- The implementation of container orchestrator services(AWS ECS) allows usage of containers and scales to accommodate load fluctuations, allowing the system to maintain performance-based service level agreements (SLAs) even when under heavy load or stress.

2. Resiliency- The use of decoupled services (using asynchronous messaging) allows the core value of the business (taking orders) to remain operational despite the potential for the services on the periphery (the services that provide notifications) to fail.

3. Infrastructure as Code (IaC) and Continuous Integration/Continuous Deployment (CI/CD) pipelines promote a standardized way to deploy code while minimizing risks and delays during the deployment process.

This research has provided significant quality evidence for the hypothesis of the research question. As stated above, the use of a microservice-based deployment solution will lead to a greater ability to perform Continuous Integration / Continuous Delivery (CI/CD) and DevOps than with a traditional monolithic architecture by being able to perform CI/CD more quickly and having a more flexible and scalable deployment solution. However, due to the complexity of managing a distributed system, an organization must have achieved a higher level of DevOps maturity to effectively manage their distributed applications.

7.2 Limitations

Despite the success of the implementation, several limitations are acknowledged:

- The Payment Gateway service uses simulated payments for both security and scope purposes, rather than directly integrating with a live payment processing provider such as Stripe.

- The services logically separate; however, all of their data resides in one shared physical RDS Instance, as would occur in large deployments (hyper-scale) where the single instance would become subject to being a ”Noisy Neighbor” — overloading that instance with requests.

- Finally, currently, the deployment is only available in one of AWS’s Regions, so it is susceptible to Regional Outages; this could also cause problems for customers located outside of that region.

7.3 Future Work

Future research could build upon this foundation by exploring:

- Integration of a Service Mesh allows you to have advanced control of how traffic flows between your applications and other services within the service mesh. Using a Service Mesh will also allow you to increase your application's security through the use of mutual TLS, gain deeper insight into your application's behavior with advanced visibility, and implement the changes needed to utilize a Service Mesh with no code changes.
- Serverless database migration is possible using Amazon Aurora Serverless, which automatically scales the database to provide true database elasticity to as much extent as your compute layer does.
- Chaos engineering provides you with the ability to automatically discover the points of failure in an application's resiliency by using services such as AWS Fault Injection Simulator to inject faults into your applications.

References

Alshammari, H. (2024). Performance evaluation of headless architectures in e-commerce applications, *Journal of Systems and Software* **189**: 111300.

Balalaie, A., Heydarnoori, A. and Jamshidi, P. (2016). Microservices architecture: Enablement and evaluation, *IEEE Software* .

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R. and Safina, L. (2022). Microservices: yesterday, today, and tomorrow, *Present and ulterior software engineering* pp. 195–216.

Evans, E. (2004). *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley.

Jaramillo, D., Nguyen, D. V. and Smart, R. (2016). Leveraging microservices architecture by using docker technology, *SoutheastCon 2016* .

Lewis, J. and Fowler, M. (2014). Microservices: a definition of this new architectural style, *MartinFowler.com* .
URL: *https://martinfowler.com/articles/microservices.html*

Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment, *Linux Journal*, Vol. 2014, p. 2.

Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems*, O'Reilly Media.

Villamizar, M. et al. (2015). Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures, *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*.

Zhou, Y. et al. (2023). A comprehensive survey on microservices architecture: State-of-the-art, challenges, and future trends, *IEEE Access* **11**: 12345–12367.