

CityMart E-Commerce Store:
**Design, Deployment, and Evaluation of a Highly Available, Scalable,
and Secure Three-Tier Architecture on AWS**

**A Technical Report Submitted in Partial Fulfilment of the Requirements for
Cloud Engineering / DevOps Architecture Studies**

Prepared by:
AKOSA SAMUEL ONYEJEKWE

Student ID:
CBALC0168

Supervisor:
Salima Rabiou/ Akinwunmi Akinrimisi

Programme:
Cloudboosta Training Programme
(Cloud Computing & DevOps)

Technologies Used:
Amazon Web Services (AWS)
CloudFront, Route 53, Application Load Balancer (ALB)
EC2 Auto Scaling Groups (ASG), Amazon RDS (MySQL)
Amazon S3, AWS Systems Manager (SSM)

Date of Submission:
April 2026

Abstract

This report presents the design, implementation, and evaluation of a production-grade three-tier cloud architecture for the CityMart online platform, deployed on Amazon Web Services (AWS). The objective of the solution is to transform a traditional retail ordering process into a scalable, secure, and highly available web-based system that enables customers to browse products, view details, and submit order requests efficiently.

The architecture is structured into three logical tiers: a presentation layer hosted on Nginx-based EC2 instances, an application layer powered by Apache and PHP APIs, and a data layer implemented using Amazon RDS MySQL in a Multi-Availability Zone configuration. The system is deployed within a custom Virtual Private Cloud (VPC) with segmented public and private subnets to enforce strict network isolation and layered security controls.

To enhance performance and global accessibility, Amazon CloudFront is integrated as a Content Delivery Network (CDN), serving as the edge entry point and caching static assets to reduce latency and origin load. Traffic routing is managed using Amazon Route 53 and secured through AWS Certificate Manager (ACM), ensuring end-to-end HTTPS communication. Application Load Balancers (ALBs) are employed to distribute traffic across multiple Availability Zones, while Auto Scaling Groups (ASGs) provide elasticity, self-healing capabilities, and horizontal scalability for both frontend and backend tiers.

The deployment model leverages Amazon S3 for centralised storage of application code, with automated instance provisioning and configuration achieved through user data scripts and AWS Systems Manager (SSM). This approach ensures consistent, repeatable deployments and minimises configuration drift.

The resulting system demonstrates strong alignment with the AWS Well-Architected Framework, achieving key objectives in security, reliability, performance efficiency, cost optimisation, and operational excellence. The CityMart platform is therefore established as a robust, cloud-native solution capable of supporting real-world workloads with high availability, scalability, and maintainability.

Contents

Abstract	2
1. Introduction.....	16
1.1 Business Scenario and Problem Statement	16
1.2 Project Objectives	16
1. Enable Digital Customer Interaction	16
2. Ensure High Availability and Reliability	16
3. Achieve Scalability and Elasticity.....	17
4. Enforce Strong Security Controls	17
5. Establish a Maintainable Deployment Model	17
6. Deliver a Production-Ready Architecture	17
1.3 Scope of the Solution.....	17
Included in Scope	17
Out of Scope	18
1.4 Overview of the CityMart Platform	18
1. Presentation Tier (Frontend Layer)	18
2. Application Tier (Backend Layer)	19
3. Data Tier (Database Layer)	19
4. Supporting Infrastructure Components	21
5. End-to-End Functional Flow	22
6. Architectural Significance.....	23
2. Solution Overview	23
2.1 High-Level Architecture Description	23
2.2 Three-Tier Architecture Model	24
Presentation Tier (Frontend Layer)	24
Application Tier (Backend Layer).....	25
Data Tier (Database Layer).....	25
2.3 Key AWS Services Used	26
Networking & Access	26
DNS & Security	26

Content Delivery.....	26
Load Balancing	26
Compute & Scaling	26
Storage & Deployment.....	26
Database	26
2.4 End-to-End Traffic Flow Overview	27
Step-by-Step Flow	27
3. Architecture Design Principles	28
3.1 High Availability (Multi-AZ Design)	28
Implementation Details	28
Outcome	29
3.2 Scalability (Auto Scaling Model).....	29
Implementation Details	29
Key Characteristics	30
Outcome	30
3.3 Security (Layered + Zero Trust Architecture)	30
Network Isolation	30
Security Group Design (5-Tier Model)	31
Additional Security Controls	33
Outcome	34
3.4 Performance Optimisation	34
CloudFront CDN Integration	34
Load Balancing	34
Efficient Request Routing	35
Compression and Optimisation	35
Outcome	35
3.5 Cost Optimisation	35
Auto Scaling Efficiency	35
Instance Selection.....	36
S3-Based Deployment	36
CloudFront Caching.....	36
Managed Services Usage	36

Network Efficiency	36
Outcome	37
4. Network Architecture (VPC Design).....	37
4.1 VPC Configuration (CIDR, DNS Settings).....	37
Design Rationale	38
4.2 Subnet Design and Tier Segmentation	38
Subnet Breakdown	38
Public Subnets	40
Private Web Subnets (Frontend Tier).....	40
Private Application Subnets (Backend Tier)	41
Private Database Subnets (Data Tier).....	41
Key Design Outcome	41
4.3 Internet Gateway Configuration.....	42
Purpose	42
Design Importance	42
4.4 NAT Gateway Architecture.....	42
Traffic Flow.....	43
Purpose	43
High Availability Design	43
4.5 Route Table Configuration	43
Public Route Table.....	44
Private Route Tables	44
Design Insight.....	44
4.6 Network Isolation Strategy	44
Traffic Flow Enforcement.....	44
Isolation Characteristics	45
Engineering Outcome	45
5. DNS and TLS Layer (Route 53 + ACM)	46
5.1 Domain Name Configuration	46
Configured Domain	46
Purpose of Domain Configuration.....	46
5.2 Route 53 Hosted Zone Setup.....	47

Hosted Zone Configuration	48
Role of Route 53	48
Key Capabilities Leveraged	48
Engineering Insight	48
5.3 Alias Record Configuration	48
Initial Configuration	48
Final Production Configuration	48
Alias Record Details	48
Why Alias Records?	49
Engineering Benefit	49
5.4 SSL/TLS Implementation with ACM	49
Certificate Configuration	49
Integration Points	50
TLS Architecture	50
Why This Design?	50
Security Benefits	50
5.5 HTTPS Enforcement Strategy	50
CloudFront Layer	51
ALB Layer	51
Combined Enforcement Model	51
Engineering Outcome	51
5.6 Evolution to Production Pattern (CloudFront Integration)	51
Initial Architecture	51
Final Production Architecture	51
Why CloudFront Was Introduced	51
Key Outcome	52
6. Content Delivery Layer (CloudFront CDN)	52
6.1 CloudFront Architecture Integration	52
6.2 Distribution Configuration	53
Key Distribution Properties	53
Behaviour 1 - Application Traffic	54
Behaviour 2 - Default Viewer Policy	54

6.3 Origin Design (ALB as Origin)	55
Origin Endpoint	55
Rationale for ALB as Origin	55
Protocol Design Decision	55
6.4 Caching Strategy (Static vs Dynamic Content)	56
Static Content (Cached)	56
Dynamic Content (Not Cached)	56
Reasoning	56
6.5 Security Features (HTTPS, Shield, WAF Readiness)	57
1. HTTPS Enforcement	57
2. AWS Shield (DDoS Protection)	57
3. Origin Protection	57
4. WAF Readiness	57
6.6 Performance Optimisation (Edge Caching, Compression)	57
1. Edge Caching	58
2. Reduced Origin Load	58
3. Automatic Compression	58
4. Global Performance Consistency	58
6.7 Cache Invalidation Strategy	58
Invalidation Configuration	58
Purpose	59
Outcome	59
7. Application Load Balancers - ALBs	59
7.1 Frontend ALB (Internet-Facing)	60
HTTP → HTTPS Redirection	60
HTTPS Listener with ACM	61
Target Group Integration	62
7.2 Backend ALB (Internal)	62
Private Routing	62
Application Isolation	63
7.3 Target Groups and Health Checks	63
Target Groups per Tier	63

Load Balancing Function	64
Health Checks	64
Integration with Auto Scaling	65
7.4 Request Routing Logic	65
End-to-End Flow	65
Layered Routing Logic	66
Key Architectural Benefits	66
8. Compute Layer (EC2, AMI, Launch Templates, ASG)	67
8.1 EC2 Instance Configuration	67
Frontend EC2 (Web Tier)	68
Backend EC2 (Application Tier)	69
Key Configuration Characteristics	69
Engineering Insight	69
8.2 Amazon Machine Images (AMI)	69
Role of AMIs in the Architecture	69
Engineering Insight	70
8.3 Launch Templates Design	70
Frontend Launch Template	70
Backend Launch Template	71
Key Capabilities	72
Engineering Insight	72
8.4 Auto Scaling Groups (Frontend & Backend)	73
Frontend ASG	73
Backend ASG	74
Configuration	75
Integration Flow	75
Key Capabilities	75
Engineering Insight	76
8.5 Scaling Behaviour and Self-Healing	76
Scaling Behaviour	76
Self-Healing Mechanism	76
Load Balancer Interaction	76

Engineering Insight	77
8.6 Multi-AZ Deployment Strategy	77
Implementation.....	77
Benefits	77
Traffic Distribution	77
Engineering Insight	77
9. Application Architecture	78
9.1 Frontend Application Structure	78
Structure and Components	78
Frontend Responsibilities.....	79
Static Asset Delivery.....	79
9.2 Backend Application Structure	80
Structure and Components	80
Backend Responsibilities	80
9.3 API Design and Endpoints	81
Core Endpoints.....	81
API Design Characteristics	81
9.4 Application Request Flow	82
Step-by-Step Flow	82
9.5 Nginx Reverse Proxy Configuration	82
Key Responsibilities	82
Core Configuration Behaviour.....	82
Reverse Proxy for API.....	83
9.6 Apache Backend Configuration	83
Core Configuration	83
Key Configuration Elements	83
Security and Headers	84
Design Considerations	84
TLS Handling.....	84
10. Application Deployment Model	84
10.1 S3-Based Code Storage	85
Design Rationale	86

Key Characteristics	86
10.2 Deployment Strategy (S3 Sync)	87
Frontend Deployment Example	87
Backend Deployment Example	87
How It Works	87
Engineering Advantages	87
10.3 User Data Scripts (Frontend & Backend)	87
Frontend User Data Script Responsibilities	87
Functional Role	88
Backend User Data Script Responsibilities	88
Functional Role	88
Key Design Principle	88
10.4 SSM-Based Deployment Execution	88
Why SSM Was Used	89
Execution Model	89
Connectivity Path	89
Engineering Benefits	89
10.5 Update and Release Strategy	89
Release Workflow	90
Deployment Options	90
Why This Strategy Works	90
Operational Characteristics	90
11. Database Layer (Amazon RDS)	91
11.1 Database Architecture Overview	91
Architectural Placement	91
11.2 Subnet Group and Private Deployment	92
Key Design Decisions	93
Engineering Rationale	94
11.3 Multi-AZ Configuration	94
Implementation Details	94
How It Works	94
Key Benefits	95

Engineering Insight	95
11.4 Database Security Model	95
1. Network-Level Security (Primary Control)	95
Traffic Enforcement Chain	96
2. Layer Isolation	96
3. Security Principles Applied	96
11.5 Connectivity and Authentication.....	96
1. Connectivity Model	96
2. Authentication Model.....	97
3. Access Control Layers	97
4. Design Implications	97
11.6 Monitoring and Observability	97
Monitoring Features Implemented.....	97
Observability Benefits	98
Alignment with AWS Well-Architected Framework	98
Engineering Insight	98
12. Systems Management and Operations	99
12.1 AWS Systems Manager (SSM)	99
▪ Functional Role of SSM in CityMart	99
▪ Architectural Placement	99
12.2 Session Manager Access Model	100
▪ How Session Manager Works	100
▪ Key Characteristics of the Access Model	100
12.3 IAM Roles and Permissions	101
▪ Required IAM Role	101
▪ What This Role Enables	101
▪ Security Benefits	101
12.4 Private Instance Management Strategy	102
▪ Core Design Principles	102
▪ How Management is Achieved	102
▪ Why NAT Gateway is Critical	102
▪ Optional Enhancement (Not Implemented)	103

Engineering Summary	103
1. Security	103
2. Reliability	103
3. Scalability.....	103
4. Operational Excellence	103
13. End-to-End System Flow	103
13.1 Full Request Lifecycle.....	104
Step-by-Step Lifecycle	104
1. Client Request Initiation	104
2. Edge Layer - CloudFront CDN	104
3. Origin Layer - Public Application Load Balancer	104
4. Presentation Layer - Frontend EC2 (Nginx)	105
5. Application Routing - Internal ALB	105
6. Application Layer - Backend EC2 (PHP API)	105
7. Data Layer - Amazon RDS MySQL	106
8. Response Flow Back to User	106
13.2 Component Interaction Sequence	107
Logical Interaction Flow	107
Interaction Characteristics.....	107
1. Layered Communication.....	107
2. Load Balancer Mediation.....	107
3. Stateless Processing	108
4. Controlled API Communication	108
13.3 Data Flow Across Tiers.....	108
1. Data Flow Direction	108
2. Presentation to Application Layer	108
3. Application to Data Layer	109
4. Data Persistence and Retrieval	109
5. Response Data Flow	109
6. Security Enforcement in Data Flow	110
14. Alignment with AWS Well-Architected Framework.....	110
14.1 Operational Excellence	111

Automated Infrastructure Provisioning.....	111
Automated Deployment Model (S3 + SSM)	111
Operational Monitoring & Observability	111
Infrastructure Management Without SSH	112
Operational Excellence Outcome	112
14.2 Security	112
Network Isolation (Core Security Principle)	112
Security Group Layering (Zero Trust Architecture).....	113
Encryption in Transit (TLS/HTTPS)	113
Edge Security with CloudFront.....	113
Database Security	113
Security Outcome.....	114
14.3 Reliability	114
Multi-AZ Deployment.....	114
Load Balancing Across AZs	114
Auto Scaling Self-Healing	114
Database High Availability	115
Health Checks	115
Reliability Outcome	115
14.4 Performance Efficiency	115
CloudFront Edge Optimisation.....	115
Load Balancing Efficiency	116
Stateless Application Design	116
Independent Tier Scaling	116
Optimised Data Access	116
Performance Outcome	116
14.5 Cost Optimisation	117
Auto Scaling (Pay-as-you-use)	117
Use of Lightweight Instance Types	117
CloudFront Cost Efficiency.....	117
S3-Based Deployment	117
Private Subnet Design	117

Right-Sized Database	117
Cost Optimisation Outcome	118
15. Benefits Achieved	118
15.1 Improved Customer Experience	118
Performance Enhancement via CloudFront	118
Consistent Access via DNS and HTTPS	119
Outcome	119
15.2 High Availability and Fault Tolerance	119
Multi-AZ Deployment.....	119
Load Balancing Across Instances	119
Auto Scaling Self-Healing Mechanism	119
Database Redundancy.....	120
Outcome	120
15.3 Scalability and Elasticity.....	120
Independent Tier Scaling	120
Elastic Compute Model	120
Load Distribution	120
CDN Offloading	121
Outcome	121
15.4 Security Improvements	121
Network Isolation	121
Security Group Chaining.....	121
Encryption and Secure Access	121
Backend and Database Protection.....	122
Operational Security.....	122
Outcome	122
15.5 Maintainability and Deployment Efficiency	122
S3-Based Deployment Model.....	122
Automated Instance Configuration	122
SSM-Based Management	122
Rolling Updates and Refresh Strategy.....	123
Immutable Infrastructure	123

Outcome	123
Engineering Summary.....	123
16. Conclusion	123
16.1 Summary of Architecture	124
16.2 Key Engineering Achievements	125
1. End-to-End Layered Architecture Design.....	125
2. High Availability Through Multi-AZ Deployment.....	125
3. Horizontal Scalability and Self-Healing Infrastructure	125
4. Secure-by-Design Network Architecture	125
5. Global Performance Optimisation via CloudFront	126
6. Production-Grade DNS and TLS Implementation.....	126
7. Automated and Consistent Deployment Model.....	126
8. Operational Excellence with SSM-Based Access.....	127
16.3 Production Readiness Assessment.....	127
1. Security Readiness	127
2. Reliability and Fault Tolerance	127
3. Scalability Readiness.....	127
4. Performance Optimisation.....	128
5. Operational and Deployment Readiness	128
6. Alignment with AWS Well-Architected Framework.....	128
Final Assessment.....	129
References	129

1. Introduction

1.1 Business Scenario and Problem Statement

CityMart is a small neighbourhood retail business that traditionally relied on manual processes, primarily phone calls and in-person visits-to manage customer orders. This approach created several operational inefficiencies, including delayed order handling, high dependency on staff availability, and limited customer convenience.

To address these challenges, CityMart required a lightweight digital solution that would enable customers to interact with the store remotely while avoiding the complexity of a full-scale e-commerce system.

The core functional requirements of the system were:

- Allow customers to **browse available products** such as rice, cooking oil, detergent, and soap
- Enable users to **view detailed product information**, including pricing and descriptions
- Provide a mechanism to **submit order requests for in-store pickup**
- Generate an **order reference number upon submission**, enabling store staff to follow up manually

Importantly, the system was intentionally designed **without online payment integration**, focusing instead on simplifying the ordering process rather than fully digitising transactions.

The key problem to solve was therefore:

How to transform a manual, phone-based ordering system into a simple, reliable, and scalable digital platform while maintaining low complexity and operational control.

1.2 Project Objectives

The primary objective of this project was to design and implement a **production-grade, cloud-native 3-tier architecture** on AWS that supports the CityMart online platform.

The specific engineering objectives included:

1. Enable Digital Customer Interaction

- Provide a web-based interface for browsing products and submitting orders
- Replace manual phone ordering with a structured, self-service workflow

2. Ensure High Availability and Reliability

- Deploy infrastructure across **multiple Availability Zones (eu-west-2a and eu-west-2b)**

- Use load balancing and redundancy to eliminate single points of failure

3. Achieve Scalability and Elasticity

- Implement **Auto Scaling Groups (ASGs)** for both frontend and backend tiers
- Allow the system to dynamically adjust to varying traffic demands

4. Enforce Strong Security Controls

- Isolate all compute resources in **private subnets**
- Apply **least-privilege security group rules** across tiers
- Ensure secure access using **HTTPS via ACM and controlled DNS routing**

5. Establish a Maintainable Deployment Model

- Store application code in **Amazon S3**
- Use **user data scripts and SSM** for automated provisioning and updates
- Avoid manual configuration drift

6. Deliver a Production-Ready Architecture

- Integrate **Route 53, CloudFront, ALB, EC2, and RDS**
- Implement a clean, layered architecture aligned with AWS best practices

These objectives collectively ensured that the solution was not only functional but also aligned with **modern cloud engineering principles**, including automation, resilience, and security.

1.3 Scope of the Solution

The solution scope was deliberately defined to balance **business needs, system simplicity, and architectural robustness**.

Included in Scope

Application Features

- Product catalog browsing
- Product detail viewing
- Order request submission
- Order confirmation via reference number

Infrastructure Components

- **Presentation Layer:** Nginx-based frontend on EC2
- **Application Layer:** Apache + PHP backend APIs
- **Data Layer:** Amazon RDS MySQL database

Cloud Architecture

- Fully implemented **3-tier architecture**

- ## Networking and Security

- ## Deployment and Operations

- ## Out of Scope

- This scoped approach ensured that the solution remained **focused, cost-effective, and aligned with the business requirement**, while still demonstrating a **fully production-grade cloud architecture**.

Page 18 of 130

- Handles user interaction and UI rendering
- Forwards API requests (/api/*) to the backend layer

The frontend is designed to be **stateless**, enabling horizontal scaling via Auto Scaling Groups.

2. Application Tier (Backend Layer)

- Hosted on **EC2 instances running Apache and PHP**
- Provides RESTful API endpoints such as:
 - /products
 - /product?id=1
 - /order
- Responsible for:
 - Business logic execution
 - Data processing
 - Communication with the database

This tier is deployed behind an **internal ALB**, ensuring it is **not directly accessible from the internet**, thereby enhancing security.

3. Data Tier (Database Layer)

- Implemented using **Amazon RDS MySQL**
- Stores application data including:
 - products
 - customers
 - orders
 - order_items
- Configured for:
 - **Multi-AZ deployment** (high availability)
 - **Private subnet isolation** (no public exposure)

This ensures **data durability, fault tolerance, and secure access control**.

Products

Search products...



Premium Rice 5kg
\$12.99

Long grain rice, 5kg bag.

View

Add



Cooking Oil 1L
\$4.50

Vegetable cooking oil, 1 litre.

View

Add



Laundry Detergent
\$8.75

Fresh scent detergent, 2kg.

View

Add



Bath Soap (Pack of 4)
\$3.20

Mild soap pack.

View

Add

[Back to products](#)



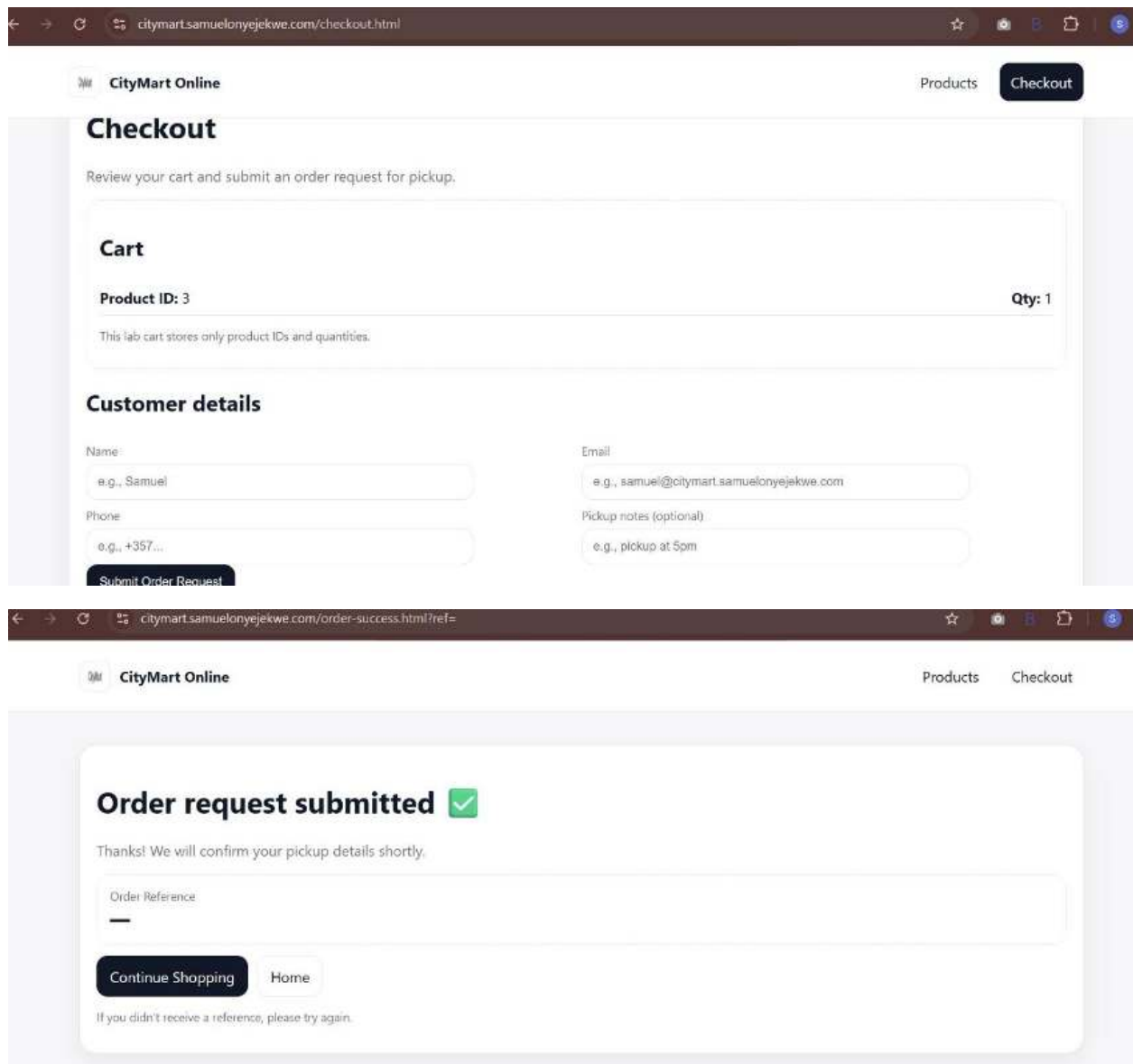
Bath Soap (Pack of 4)

\$3.20

Mild soap pack.

1

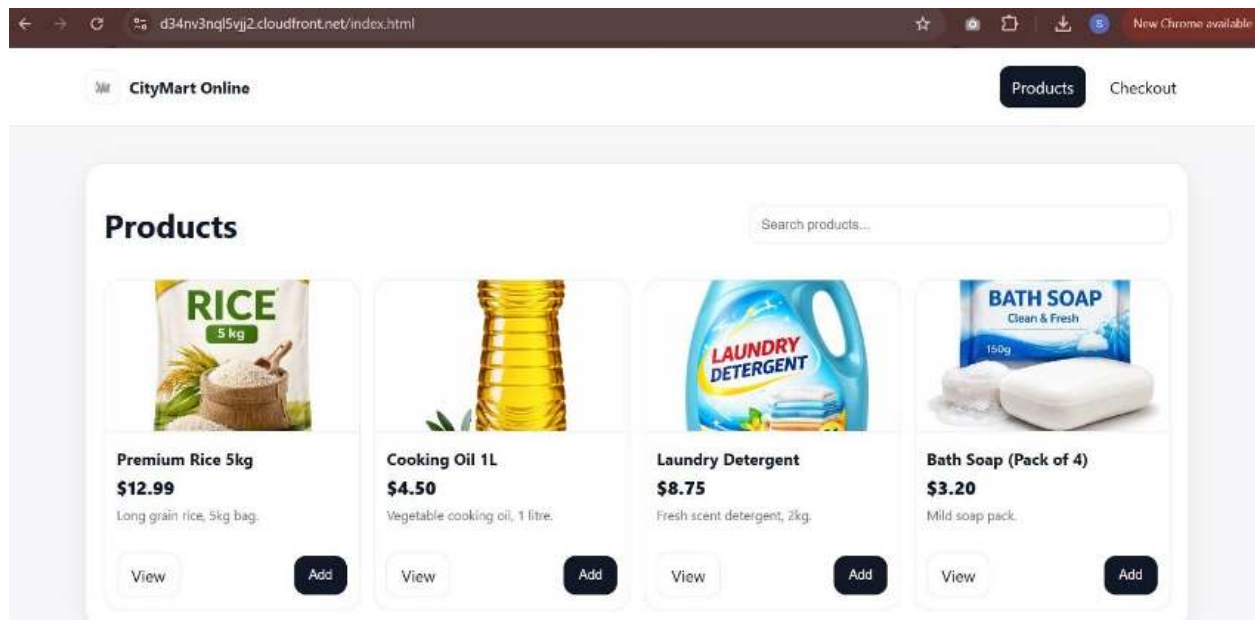
Add to Cart



4. Supporting Infrastructure Components

CloudFront (CDN Layer)

- Acts as the **global entry point**
- Caches static content at edge locations
- Reduces latency and improves performance



Route 53 (DNS Layer)

- Maps the custom domain:
`citymart.samuelonyejekwe.com`
to the application

Application Load Balancers

- **Public ALB:** Handles incoming HTTPS traffic
- **Internal ALB:** Routes requests to backend services

Auto Scaling Groups

- Ensure:
 - dynamic scaling
 - self-healing
 - high availability

5. End-to-End Functional Flow

The platform supports the following user journey:

1. User accesses the website via the custom domain
2. DNS resolves via Route 53 and routes traffic through CloudFront
3. CloudFront forwards requests to the public ALB
4. Frontend servers serve static content
5. API requests are routed to backend services
6. Backend processes requests and interacts with the database
7. Response is returned to the user with an order confirmation

6. Architectural Significance

The CityMart platform demonstrates:

- A **production-grade 3-tier architecture**
- Strong alignment with **AWS Well-Architected principles**
- Effective use of:
 - **network isolation**
 - **load balancing**
 - **auto scaling**
 - **secure access patterns**

Ultimately, the system successfully transforms a manual retail workflow into a **modern, cloud-enabled digital platform**, while maintaining simplicity, security, and scalability.

2. Solution Overview

2.1 High-Level Architecture Description

The CityMart platform was implemented using a **production-grade three-tier architecture on AWS**, enhanced with **content delivery, load balancing, and automated scaling capabilities** to ensure high availability, security, and performance.

At a high level, the architecture follows a **layered, service-oriented design**, where each tier is independently deployed, secured, and scaled. The system is structured to enforce **strict separation of concerns**, ensuring that presentation, business logic, and data storage operate in isolated environments while communicating through controlled interfaces.

The complete architecture integrates the following major components:

- **Amazon CloudFront** as a global content delivery and edge security layer
- **Amazon Route 53** for DNS resolution and domain routing
- **Application Load Balancers (ALBs)** for traffic distribution
- **Amazon EC2 instances** organised into Auto Scaling Groups for compute
- **Amazon RDS (MySQL)** for persistent data storage
- **Amazon S3** for application code storage and deployment

The architecture is deployed within a **custom Virtual Private Cloud (VPC)** spanning **two Availability Zones (eu-west-2a and eu-west-2b)** to ensure fault tolerance and resilience.

Traffic enters the system through a **custom domain configured in Route 53**, is routed via **CloudFront**, and then flows through multiple controlled layers before reaching the database.

This design ensures:

- **Global accessibility with low latency**
- **No direct exposure of compute or database resources**
- **Independent scaling of frontend and backend tiers**
- **Secure, encrypted communication at the edge**

The architecture represents a **fully production-ready deployment model**, adhering to AWS best practices for **security, scalability, and reliability**.

2.2 Three-Tier Architecture Model

The CityMart system is logically divided into three distinct layers:

Presentation Tier (Frontend Layer)

The presentation tier is responsible for handling all **user interactions and delivering the web interface**.

It consists of:

- **CloudFront CDN** (edge entry point)
- **Public Application Load Balancer (ALB)**
- **Frontend EC2 instances running Nginx**
- Static frontend assets (HTML, CSS, JavaScript)

CloudFront serves as the **global entry point**, caching static content such as:

- `/assets/js/*`
- `/assets/css/*`
- `/assets/img/*`

This reduces latency and improves user experience by serving content from the **nearest edge location**.

The frontend EC2 instances:

- Serve static web pages (e.g., `index.html`, `products.html`)
- Act as a **reverse proxy**, forwarding API requests (`/api/*`) to the backend tier
- Operate in **private subnets**, ensuring no direct internet access

All incoming traffic must pass through the **public ALB**, which enforces HTTPS and distributes traffic across multiple instances for high availability.

Application Tier (Backend Layer)

The application tier implements the **business logic and API services** of the CityMart platform.

It consists of:

- **Internal Application Load Balancer**
- **Backend EC2 instances running Apache and PHP**
- REST API endpoints (e.g., /api/products, /api/order)

This tier is fully isolated within **private subnets**, and is **not accessible from the internet**.

Key characteristics:

- Only the **frontend tier can communicate with the backend**, enforced via security groups
- Requests are routed through the **internal ALB**, which distributes traffic across backend instances
- Backend services handle:
 - Product retrieval
 - Order processing
 - Database interactions

This design ensures **strict internal communication boundaries**, preventing unauthorised access and enforcing a **zero-trust architecture pattern**.

Data Tier (Database Layer)

The data tier is responsible for **persistent storage and data management**.

It consists of:

- **Amazon RDS (MySQL) instance**
- **Multi-AZ deployment across private database subnets**

The database stores structured data including:

- Products
- Customers
- Orders
- Order items

Key architectural features:

- **Deployed in private subnets only** (no public access)
- Accessible exclusively from backend EC2 instances via **security group rules**

- Configured with **Multi-AZ failover**, ensuring high availability and automatic recovery in case of failure

This ensures:

- **Data durability**
- **High availability**
- **Strict access control**

2.3 Key AWS Services Used

The CityMart architecture leverages multiple AWS services, each fulfilling a specific role:

Networking & Access

- **Amazon VPC** - Provides isolated network environment
- **Subnets (Public & Private)** - Enforce tier separation
- **Internet Gateway (IGW)** - Enables public connectivity
- **NAT Gateways** - Allow outbound internet access for private resources

DNS & Security

- **Amazon Route 53** - Domain name resolution and routing
- **AWS Certificate Manager (ACM)** - SSL/TLS certificate management
- **HTTPS enforcement** via ALB and CloudFront

Content Delivery

- **Amazon CloudFront** - Global CDN, caching, and edge security layer

Load Balancing

- **Public ALB** - Internet-facing traffic distribution
- **Internal ALB** - Backend service routing

Compute & Scaling

- **Amazon EC2** - Hosts frontend and backend services
- **Auto Scaling Groups (ASG)** - Enable dynamic scaling and self-healing
- **Launch Templates** - Ensure consistent instance configuration

Storage & Deployment

- **Amazon S3** - Stores application code (frontend and backend)
- **AWS Systems Manager (SSM)** - Enables secure instance management without SSH

Database

- **Amazon RDS (MySQL)** - Managed relational database with Multi-AZ support

Each service is integrated to create a **cohesive, scalable, and secure architecture**, with clearly defined responsibilities.

2.4 End-to-End Traffic Flow Overview

The CityMart architecture follows a **well-defined, layered request-response flow**, ensuring that all interactions pass through controlled entry points and secure channels.

Step-by-Step Flow

1. User Request Initiation

The user accesses the application via the custom domain:
`https://citymart.samuelonyejekwe.com`

2. DNS Resolution (Route 53)

Route 53 resolves the domain name and routes traffic to **CloudFront**.

3. CloudFront (Edge Layer)

- Terminates HTTPS connections
- Serves cached static content (if available)
- Forwards requests to the origin (Public ALB)

4. Public Application Load Balancer (Frontend ALB)

- Receives traffic from CloudFront
- Enforces HTTPS (HTTP → HTTPS redirect)
- Distributes requests across frontend EC2 instances

5. Frontend EC2 Instances (Nginx)

- Serve static web content
- Process user interface requests
- Forward API calls (`/api/*`) to the backend via internal ALB

6. Internal Application Load Balancer

- Routes API requests from frontend to backend EC2 instances
- Ensures backend availability and load distribution

7. Backend EC2 Instances (Application Layer)

- Execute business logic
- Process API requests
- Query the database

8. Amazon RDS (Database Layer)

- Stores and retrieves application data
- Returns results to backend

9. Response Path

The response flows back through the same path:

Backend → Internal ALB → Frontend → Public ALB → CloudFront → User

This entire flow is explicitly defined in our architecture:

User → CloudFront → Public ALB → Frontend EC2 → Internal ALB → Backend EC2 → RDS

This solution demonstrates a **well-architected, production-grade system** with:

- **Layered traffic control (CloudFront → ALB → EC2)**
- **Strict tier isolation (Web → App → DB)**
- **Horizontal scalability via ASGs**
- **Edge optimisation via CloudFront**
- **Zero direct exposure of critical resources**

In simple terms:

The architecture transforms a basic retail workflow into a **secure, scalable, highly available cloud-native application**, aligned with AWS best practices.

3. Architecture Design Principles

The CityMart 3-tier architecture was designed and implemented using core cloud engineering principles aligned with AWS best practices. These principles ensure that the system is **highly available, scalable, secure, performant, and cost-efficient**, while maintaining operational simplicity and production readiness.

3.1 High Availability (Multi-AZ Design)

High availability in the CityMart architecture is achieved through a **multi-Availability Zone (Multi-AZ) deployment strategy**, ensuring that the system remains operational even in the event of infrastructure failure within a single zone.

The architecture is deployed in the **eu-west-2 (London) region**, utilising **two Availability Zones: eu-west-2a and eu-west-2b**. Each critical component is distributed across these zones to eliminate single points of failure.

Implementation Details

- **Frontend Layer**
 - EC2 instances (Nginx) are deployed across two private subnets (one per AZ)

- Public ALB spans both AZs and distributes traffic automatically
- **Backend Layer**
 - Backend EC2 instances (Apache + PHP) are deployed across two private subnets
 - Internal ALB operates across both AZs for redundancy
- **Database Layer**
 - Amazon RDS is configured with **Multi-AZ deployment**
 - Primary database runs in one AZ, with a standby replica in another AZ for automatic failover
- **Network Layer**
 - Two NAT Gateways (one per AZ) ensure outbound connectivity redundancy
 - Public and private subnets are distributed evenly across AZs

Outcome

- If one AZ fails:
 - Load balancers automatically route traffic to healthy instances in the other AZ
 - Auto Scaling replaces failed instances
 - RDS performs automatic failover without application disruption

This ensures **fault tolerance, service continuity, and resilience**, which are essential for production-grade systems.

3.2 Scalability (Auto Scaling Model)

The CityMart architecture is designed to support **horizontal scalability**, allowing it to dynamically adjust capacity based on workload demand.

This is achieved through **Auto Scaling Groups (ASGs)** for both the frontend and backend tiers, combined with load balancer integration.

Implementation Details

- **Frontend Auto Scaling Group**
 - Maintains baseline capacity (e.g., 2 EC2 instances)
 - Scales based on demand
 - Instances are automatically registered with the frontend target group
- **Backend Auto Scaling Group**
 - Independently scales backend API servers
 - Handles application logic and database interactions
- **Launch Templates**

- Ensure consistent configuration for all instances
- Include:
 - OS configuration
 - Application deployment (via S3 sync)
 - Service startup scripts
- **Target Group Integration**
 - Newly launched instances are automatically registered
 - Unhealthy instances are removed and replaced

ASG behaviour ensures:

- Automatic provisioning
- Self-healing
- Elastic capacity management

Key Characteristics

- **Stateless Architecture**
 - Instances do not store persistent state
 - Allows safe replacement and scaling
- **Independent Tier Scaling**
 - Frontend and backend scale separately
 - Prevents over-provisioning

Outcome

- System can handle:
 - Traffic spikes (e.g., multiple users browsing products)
 - Sudden load increases

This ensures **elastic scalability, operational efficiency, and high performance under load.**

3.3 Security (Layered + Zero Trust Architecture)

Security in the CityMart architecture is implemented using a **layered defense model** combined with **zero-trust principles**, ensuring strict control over all traffic flows.

Network Isolation

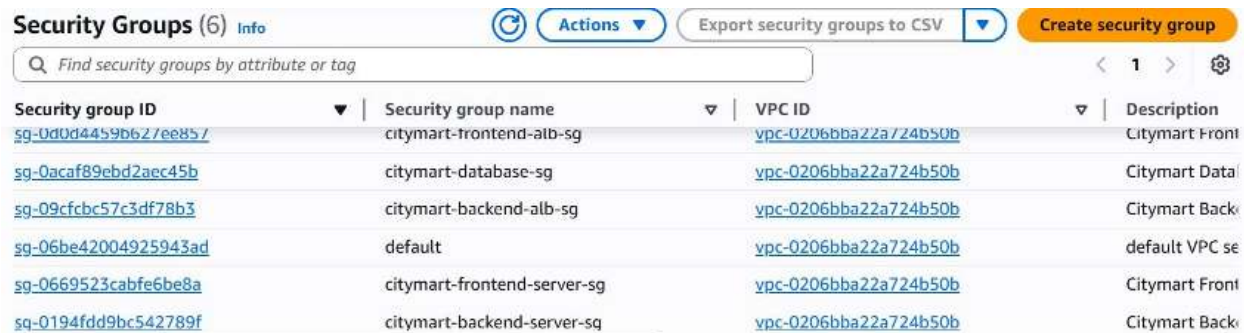
- All EC2 instances (frontend and backend) are deployed in **private subnets**
- Only the **public ALB is exposed to the internet**
- Backend and database tiers are completely isolated

Traffic flow is strictly controlled:

Internet → Public ALB → Frontend EC2 → Internal ALB → Backend EC2 → RDS

Security Group Design (5-Tier Model)

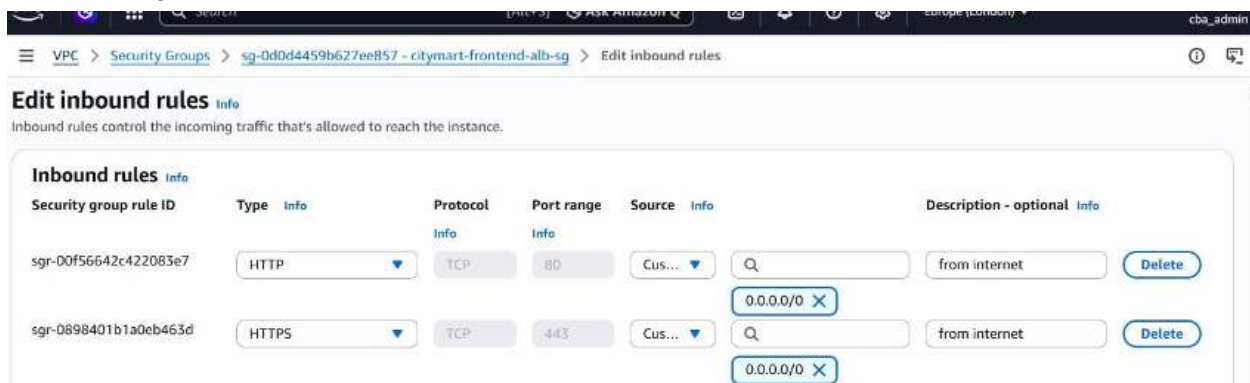
Five security groups enforce **least-privilege access**:



Security Groups (6) Info				Actions	Export security groups to CSV	Create security group
Find security groups by attribute or tag						
Security group ID	Security group name	VPC ID	Description			
sg-0d0d4459b627ee857	citymart-frontend-alb-sg	vpc-0206bba22a724b50b	Citymart Front			
sg-0acaf89ebd2aec45b	citymart-database-sg	vpc-0206bba22a724b50b	Citymart Data			
sg-09cfc57c3df78b3	citymart-backend-alb-sg	vpc-0206bba22a724b50b	Citymart Back			
sg-06be42004925943ad	default	vpc-0206bba22a724b50b	default VPC se			
sg-0669523cabfe6be8a	citymart-frontend-server-sg	vpc-0206bba22a724b50b	Citymart Front			
sg-0194fdd9bc542789f	citymart-backend-server-sg	vpc-0206bba22a724b50b	Citymart Back			

1. Frontend ALB SG

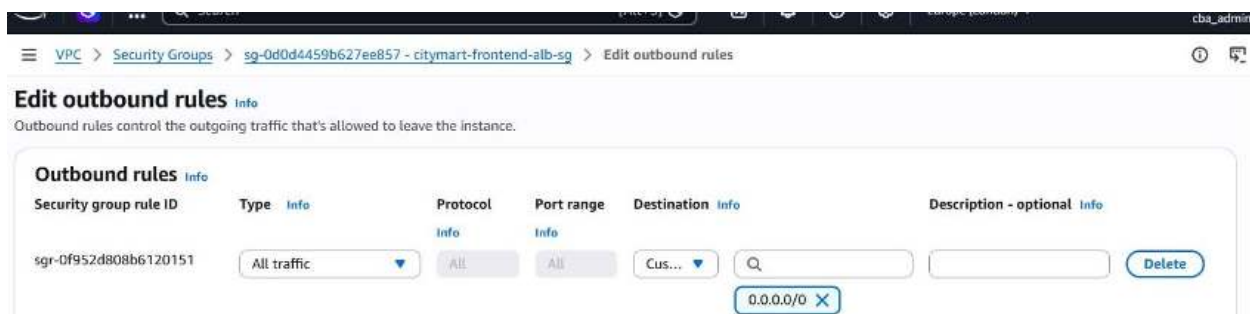
- Allows HTTP/HTTPS from internet



Edit inbound rules [Info](#)

Inbound rules control the incoming traffic that's allowed to reach the instance.

Security group rule ID	Type	Protocol	Port range	Source	Description - optional	
sgr-00f56642c422083e7	HTTP	TCP	80	Cus...	from internet	Delete
sgr-0898401b1a0cb463d	HTTPS	TCP	443	Cus...	from internet	Delete



Edit outbound rules [Info](#)

Outbound rules control the outgoing traffic that's allowed to leave the instance.

Security group rule ID	Type	Protocol	Port range	Destination	Description - optional	
sgr-0f952d808b6120151	All traffic	All	All	Cus...		Delete

2. Frontend EC2 SG

- Accepts traffic only from ALB

VPC > Security Groups > sg-0669523cabfe6be8a - citymart-frontend-server-sg > Edit inbound rules

Edit inbound rules Info

Inbound rules control the incoming traffic that's allowed to reach the instance.

Security group rule ID	Type Info	Protocol Info	Port range Info	Source Info	Description - optional Info	
sgr-0348f829ef0c0de3d	HTTP	TCP	80	Cus...	from frontend load balancer	Delete

sg-0d0d4459b627ee857

VPC > Security Groups > sg-0669523cabfe6be8a - citymart-frontend-server-sg > Edit outbound rules

Edit outbound rules Info

Outbound rules control the outgoing traffic that's allowed to leave the instance.

Security group rule ID	Type Info	Protocol Info	Port range Info	Destination Info	Description - optional Info	
sgr-088b0c259cf605b43	All traffic	All	All	Cus...		Delete

0.0.0.0/0

3. Backend ALB SG

- Accepts traffic only from frontend servers

VPC > Security Groups > sg-09cfc57c3df78b3 - citymart-backend-alb-sg > Edit inbound rules

Edit inbound rules Info

Inbound rules control the incoming traffic that's allowed to reach the instance.

Security group rule ID	Type Info	Protocol Info	Port range Info	Source Info	Description - optional Info	
sgr-029cb5e6adec3aaef	HTTP	TCP	80	Cus...	from frontend servers	Delete

sg-0669523cabfe6be8a

Add rule

Cancel Preview changes Save rules

VPC > Security Groups > sg-09cfc57c3df78b3 - citymart-backend-alb-sg > Edit outbound rules

Edit outbound rules Info

Outbound rules control the outgoing traffic that's allowed to leave the instance.

Security group rule ID	Type Info	Protocol Info	Port range Info	Destination Info	Description - optional Info	
sgr-0b385c2541272972e	All traffic	All	All	Cus...		Delete

0.0.0.0/0

4. Backend EC2 SG

- Accepts traffic only from backend ALB

VPC > Security Groups > sg-0194fdd9bc542789f - citymart-backend-server-sg > Edit inbound rules

Edit inbound rules

Inbound rules control the incoming traffic that's allowed to reach the instance.

Security group rule ID	Type	Protocol	Port range	Source	Description - optional
sg-0fcc72341b785a0f3	HTTP	TCP	80	Cus...	from backend alb

sg-09cfbc57c3df78b3

VPC > Security Groups > sg-0194fdd9bc542789f - citymart-backend-server-sg > Edit outbound rules

Edit outbound rules

Outbound rules control the outgoing traffic that's allowed to leave the instance.

Security group rule ID	Type	Protocol	Port range	Destination	Description - optional
sg-01f3d035e6d06043e	All traffic	All	All	Cus...	

0.0.0.0/0

5. Database SG

- Accepts traffic only from backend EC2 (port 3306)

VPC > Security Groups > sg-0acaf89ebd2aec45b - citymart-database-sg > Edit inbound rules

Edit inbound rules

Inbound rules control the incoming traffic that's allowed to reach the instance.

Security group rule ID	Type	Protocol	Port range	Source	Description - optional
sg-06a85a63de66891e1	MYSQL/Aurora	TCP	3306	Cus...	from backend server

sg-0194fdd9bc542789f

VPC > Security Groups > sg-0acaf89ebd2aec45b - citymart-database-sg > Edit outbound rules

Edit outbound rules

Outbound rules control the outgoing traffic that's allowed to leave the instance.

Security group rule ID	Type	Protocol	Port range	Destination	Description - optional
sg-0bc466bc089bfc8e1	All traffic	All	All	Cus...	

0.0.0.0/0

This enforces strict tier-to-tier communication

Additional Security Controls

- HTTPS Enforcement

- TLS termination via ACM at ALB and CloudFront
- **No Direct SSH Access**
 - Managed via AWS Systems Manager (SSM)
- **Private Database Deployment**
 - No public access to RDS
- **NAT-Based Outbound Access**
 - Controlled internet access for private instances

Outcome

- Zero direct exposure of backend and database
- Strict access control at every layer
- Reduced attack surface

This results in a **secure, production-grade, zero-trust architecture**.

3.4 Performance Optimisation

Performance optimisation in the CityMart architecture is achieved through a combination of **content delivery optimisation, load distribution, and efficient request routing**.

CloudFront CDN Integration

CloudFront acts as a **global edge delivery layer**:

- Caches static content:
 - /assets/js/*
 - /assets/css/*
 - /assets/img/*
- Serves content from nearest edge location
- Reduces latency and improves page load times

CloudFront also reduces load on origin servers

Load Balancing

- **Frontend ALB**
 - Distributes traffic across frontend EC2 instances

- **Backend ALB**
 - Routes API requests efficiently
- Health checks ensure traffic is only sent to healthy instances

Efficient Request Routing

- Nginx:
 - Serves static content directly
 - Proxies /api/* requests to backend
- Backend:
 - Handles business logic and database interactions

Compression and Optimisation

- CloudFront supports:
 - Gzip
 - Brotli compression
- Reduces payload size and improves response time

Outcome

- Reduced latency for global users
- Efficient backend utilisation
- Faster content delivery

Ensures **high-performance, low-latency user experience**.

3.5 Cost Optimisation

The CityMart architecture incorporates several strategies to ensure **efficient resource utilisation and cost control**.

Auto Scaling Efficiency

- Resources scale **only when needed**
- During low traffic:

- Instances scale down → reduced cost

Instance Selection

- Use of **t3.micro instances**
 - Cost-effective for lightweight workloads
 - Burstable performance model

Optimised for cost vs performance balance

S3-Based Deployment

- Application code stored in S3
- Eliminates need for:
 - Dedicated build servers
 - Persistent storage on EC2

Efficient and low-cost deployment model

CloudFront Caching

- Reduces requests to:
 - ALB
 - EC2
- Lower compute usage → reduced cost

Managed Services Usage

- RDS handles:
 - Backups
 - Failover
 - Maintenance
- Reduces operational overhead

Network Efficiency

- Private subnets reduce exposure → lower risk costs
- NAT Gateway ensures controlled outbound access

Outcome

- Pay-as-you-go model
- Reduced idle resources
- Optimised infrastructure usage

Achieves **cost-efficient, scalable cloud operation**.

The CityMart architecture successfully applies:

- **High Availability** → Multi-AZ + failover
- **Scalability** → Auto Scaling + stateless design
- **Security** → Layered + zero trust
- **Performance** → CloudFront + ALB + optimised routing
- **Cost Optimisation** → Elastic resources + managed services

This results in a **well-architected, production-grade cloud system aligned with AWS best practices**.

4. Network Architecture (VPC Design)

The CityMart platform is deployed within a **custom Amazon Virtual Private Cloud (VPC)**, which forms the foundational network boundary for all application components. The VPC design implements **strict tier isolation, controlled routing, and high availability across multiple Availability Zones**, ensuring a secure and scalable infrastructure.

4.1 VPC Configuration (CIDR, DNS Settings)

The VPC was configured with the following parameters:

- **VPC Name:** citymart-3-tier-architecture-vpc
- **CIDR Block:** 10.0.0.0/16
- **Region:** eu-west-2 (London)
- **DNS Resolution:** Enabled
- **DNS Hostnames:** Enabled

This CIDR range provides a total of **65,536 private IP addresses**, allowing sufficient address space to segment the architecture into multiple tiers without overlap.

Design Rationale

- The large CIDR block enables **future scalability** and structured subnet allocation.
- Enabling DNS resolution and hostnames allows:
 - Internal service discovery (e.g., EC2 ↔ RDS)
 - Integration with AWS-managed services such as ALB and RDS endpoints.

4.2 Subnet Design and Tier Segmentation

To enforce **network isolation and high availability**, the VPC was divided into **8 subnets across two Availability Zones**:

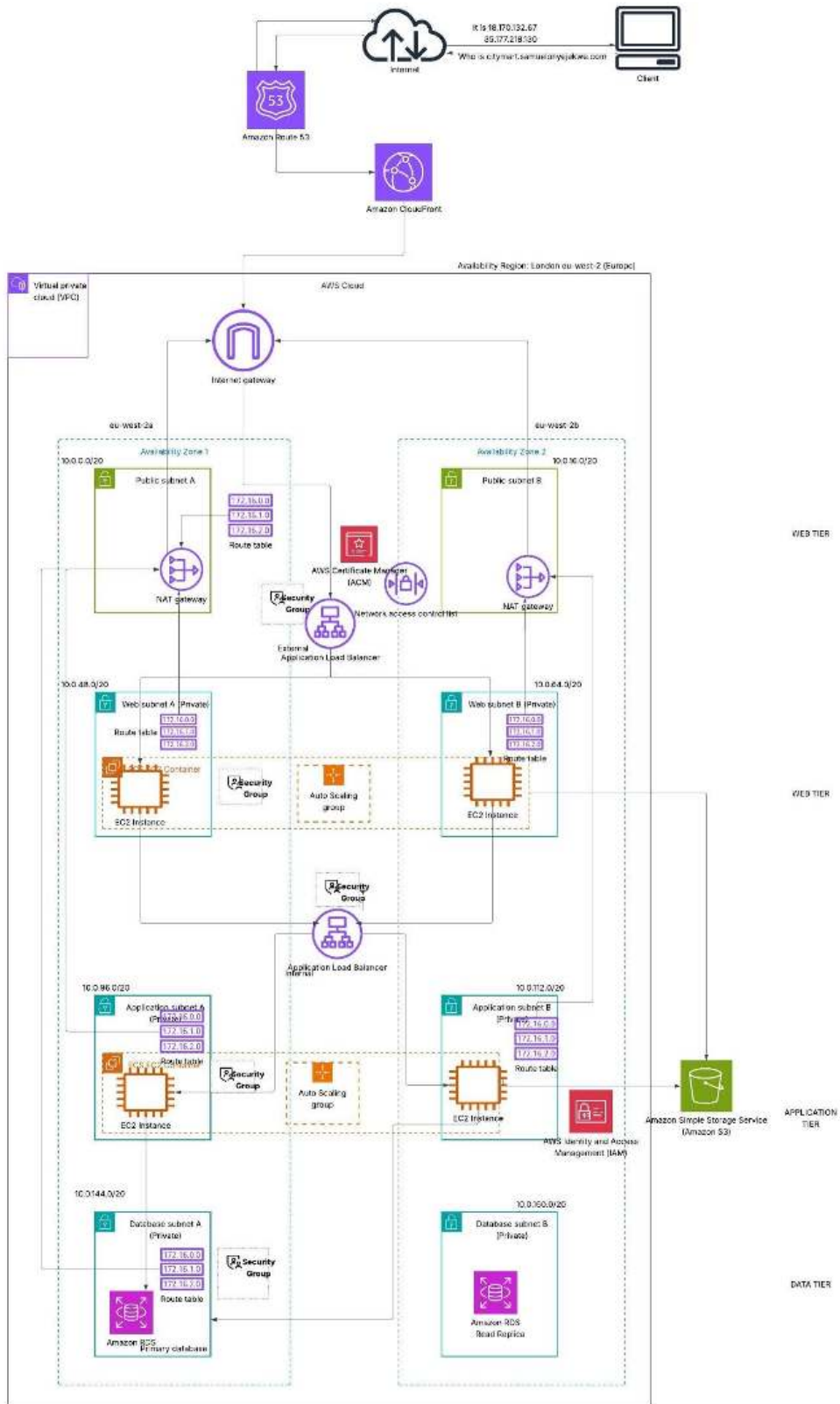
- **eu-west-2a**
- **eu-west-2b**

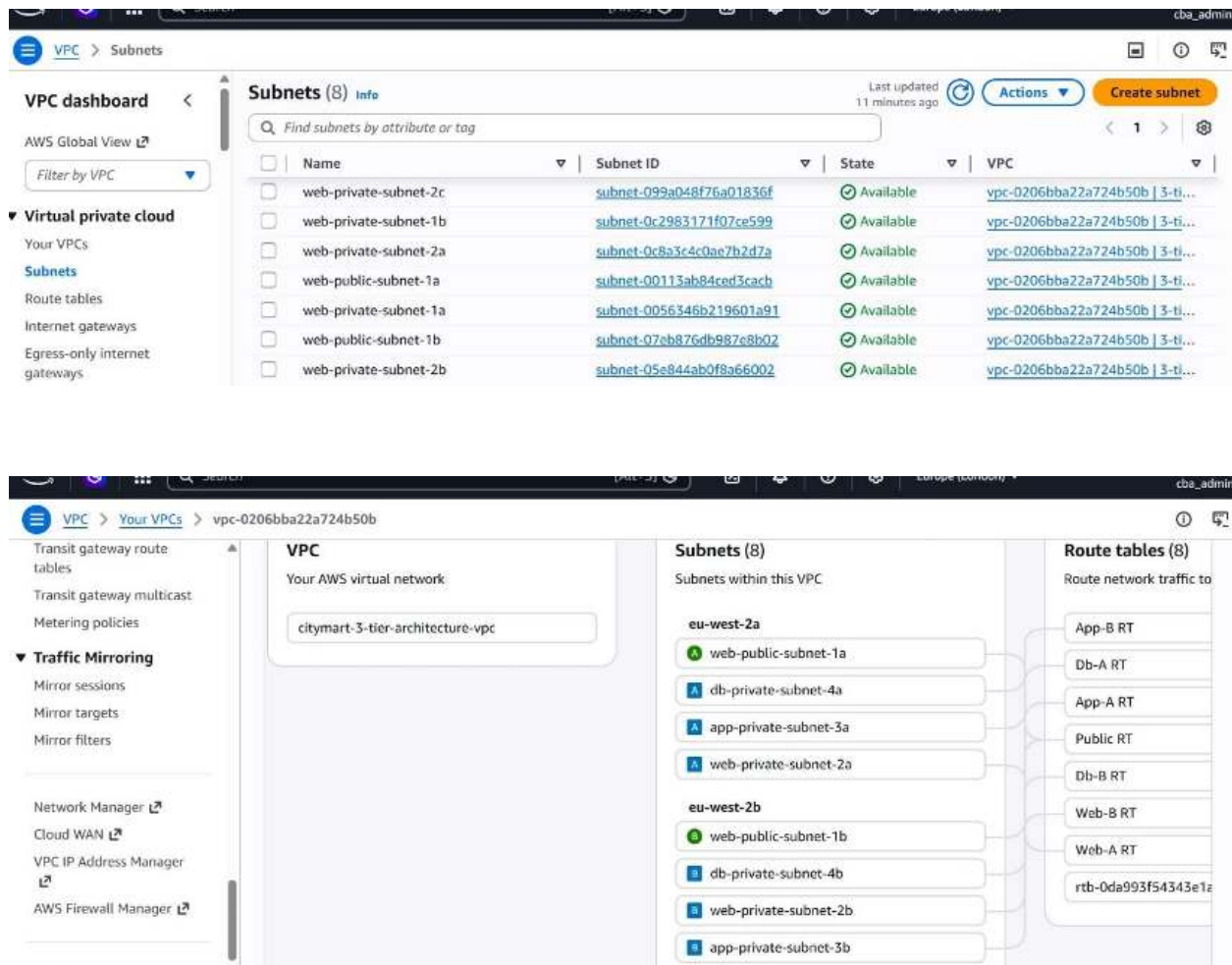
Subnet Breakdown

Tier	Subnet	CIDR	AZ	Purpose
Public	web-public-subnet-1a	10.0.0.0/20	2a	ALB + NAT
Public	web-public-subnet-1b	10.0.16.0/20	2b	ALB + NAT
Web	web-private-subnet-2a	10.0.48.0/20	2a	Frontend EC2
Web	web-private-subnet-2b	10.0.64.0/20	2b	Frontend EC2
App	app-private-subnet-3a	10.0.96.0/20	2a	Backend EC2
App	app-private-subnet-3b	10.0.112.0/20	2b	Backend EC2
DB	db-private-subnet-4a	10.0.144.0/20	2a	RDS
DB	db-private-subnet-4b	10.0.160.0/20	2b	RDS

(Derived directly from our subnet layout table and Architecture diagram)

CityMart 3-Tier Architecture





Public Subnets

These subnets host:

- **Public Application Load Balancer**
- **NAT Gateways**

Purpose:

- Serve as the **entry point for internet traffic**
- Provide outbound internet access for private resources

Private Web Subnets (Frontend Tier)

These subnets host:

- **Nginx-based frontend EC2 instances**

Characteristics:

- No direct internet access
- Only accessible via **Public ALB**

Function:

- Serve static frontend content
- Forward /api requests to backend layer

Private Application Subnets (Backend Tier)

These subnets host:

- **Backend EC2 instances (Apache + PHP)**

Characteristics:

- Fully isolated from the internet
- Accessible only via **Internal ALB**

Function:

- Execute business logic
- Process API requests

Private Database Subnets (Data Tier)

These subnets host:

- **Amazon RDS (MySQL / Aurora)**

Characteristics:

- No public IP
- Accessible only from backend tier

Function:

- Store persistent application data (products, orders, customers)

Key Design Outcome

This segmentation achieves:

- **Clear separation of concerns**
- **Strict traffic control between tiers**
- **Multi-AZ redundancy for each layer**

4.3 Internet Gateway Configuration

An **Internet Gateway (IGW)** was created and attached to the VPC:

- **IGW ID:** igw-004e50225c0fdaf27
- **State:** Attached

Purpose

The Internet Gateway enables:

- Inbound internet traffic to:
 - Public ALB
- Outbound internet access from:
 - NAT Gateways

Design Importance

Without the IGW:

- Public-facing services would not be reachable
- NAT Gateways would not function



4.4 NAT Gateway Architecture

Two NAT Gateways were deployed:

- **One per public subnet (per AZ)**
- Each associated with an **Elastic IP**

Traffic Flow

Private EC2



Private Route Table



NAT Gateway



Internet Gateway



Internet

Purpose

NAT Gateways enable private resources to:

- Install packages
- Access S3 (application code)
- Communicate with AWS services (SSM, updates)

High Availability Design

- Deploying **2 NAT Gateways (one per AZ)** eliminates single points of failure
- Ensures continued outbound connectivity if one AZ fails

Your VPCs	nat-gateway-1	nat-063a3abd490897939	Public	✓ Available	–	Zonal
Subnets	nat-gateway-2	nat-000d6bbbf11c61d75	Public	✓ Available	–	Zonal

4.5 Route Table Configuration

A total of **7 route tables** were configured:

- 1 per private subnet (6), and 1 for the 2 public subnets (fine-grained control)

Name	Route table ID	Explicit subnet associ...	Edge associations	Main
Web-B RT	rtb-02811d6235299fe5a	subnet-0c2983171f07ce...	–	No
Web-A RT	rtb-0e1f3d735b2473b46	subnet-0056346b219601...	–	No
Public RT	rtb-07cc673de7c73877d	2 subnets	–	No
Db-B RT	rtb-0e0e7ff71fe51b548	subnet-099a048f76a018...	–	No
Db-A RT	rtb-08f86c30b242a2983	subnet-0bcc97ac1cbd99f...	–	No
App-B RT	rtb-0d8d3f5db11160c37	subnet-05e844ab0f8a66...	–	No
App-A RT	rtb-0245813e6af03f594	subnet-0c8a3c4c0ae7b2...	–	No

Public Route Table

- **Destination:** 0.0.0.0/0
- **Target:** Internet Gateway

Used by:

- Public ALB
- NAT Gateways

Private Route Tables

- **Destination:** 0.0.0.0/0
- **Target:** NAT Gateway

Used by:

- Frontend EC2
- Backend EC2
- RDS (indirectly for updates if needed)

Design Insight

- Public subnets → direct internet access
- Private subnets → outbound-only via NAT

This enforces **controlled internet exposure**

4.6 Network Isolation Strategy

The CityMart architecture implements a **strict layered network isolation model**, ensuring that each tier communicates only with the tier directly above or below it.

Traffic Flow Enforcement

Internet
↓
Public ALB
↓
Frontend EC2 (private)
↓
Internal ALB
↓
Backend EC2 (private)

↓
RDS (private)

Isolation Characteristics

1. No Direct Internet Access

- Frontend, backend, and database are all in **private subnets**
- Only ALB is publicly exposed

2. Tier-to-Tier Communication Only

- Frontend → Backend via Internal ALB
- Backend → Database via SG rules

3. Controlled Entry Point

- All inbound traffic must pass through:
 - Public ALB (and CloudFront in final architecture)

4. Reduced Attack Surface

- No SSH exposure
- No public IPs on EC2 or DB

5. Zero Trust Architecture Pattern

Each layer:

- Accepts traffic **only from a specific upstream component**
- Rejects all other sources

Engineering Outcome

This network design achieves:

- **High Security** → No lateral or direct exposure
- **High Availability** → Multi-AZ deployment
- **Scalability** → Independent tier scaling
- **Operational Control** → Predictable traffic paths

The VPC architecture for CityMart is a **production-grade, multi-AZ, segmented network design** that:

- Uses **CIDR planning for scalability**
- Implements **public/private subnet separation**
- Enforces **strict tier isolation**
- Leverages **IGW + NAT for controlled connectivity**
- Uses **route tables for precise traffic control**

This is exactly aligned with **AWS Well-Architected best practices for security, reliability, and scalability**

5. DNS and TLS Layer (Route 53 + ACM)

The DNS and TLS layer in the CityMart architecture forms the **customer access layer**, enabling secure, reliable, and production-grade entry into the application. This layer integrates **Amazon Route 53 for domain resolution**, **AWS Certificate Manager (ACM) for TLS encryption**, and **CloudFront for global delivery**, creating a secure and scalable front door to the system.

The final production pattern implemented is:

User → Route 53 → CloudFront → Application Load Balancer → Application Stack

This design ensures **secure communication, high availability, and optimal performance** across all user locations.

5.1 Domain Name Configuration

A custom domain was configured for the CityMart application to provide a **user-friendly and production-ready access point**.

Configured Domain

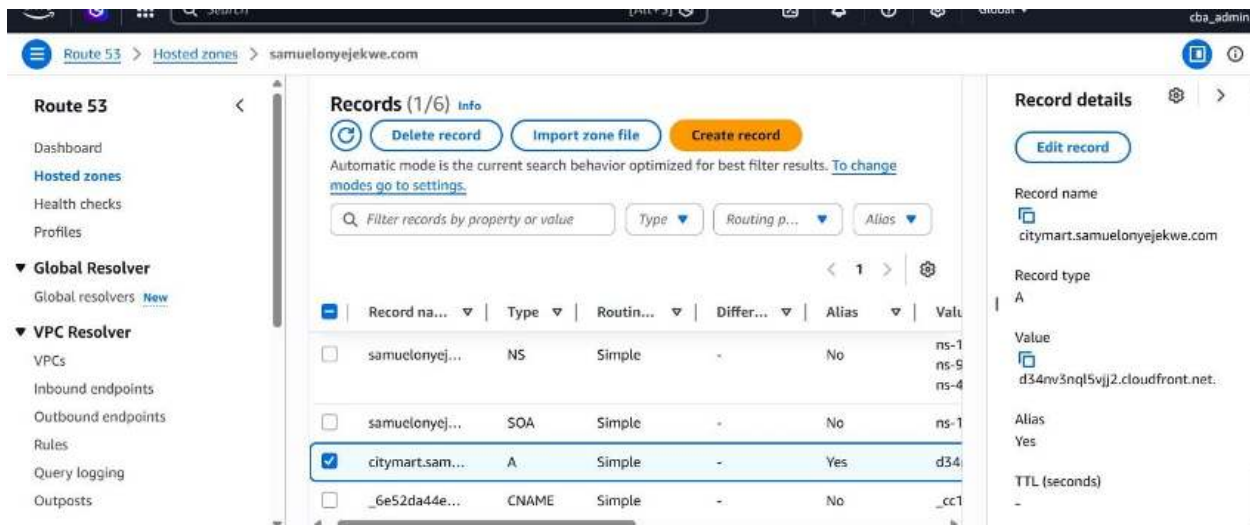
- `citymart.samuelonyejekwe.com`

This domain represents the **public-facing endpoint** of the application and abstracts the underlying AWS infrastructure from end users.

Purpose of Domain Configuration

- Provide a **consistent entry point** for customers
- Enable **HTTPS-secured communication**
- Allow integration with AWS services such as:

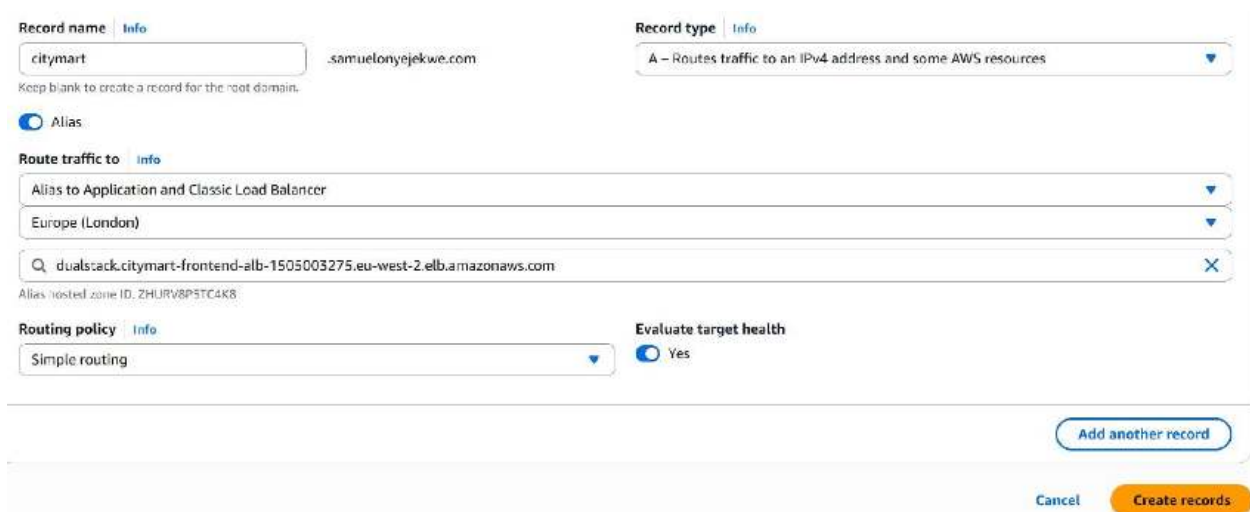
- Route 53 (DNS resolution)
- ACM (certificate management)
- CloudFront (content delivery)



From the architecture:

Customers access the system via the domain, which is resolved and routed through AWS services before reaching the application stack

This aligns with production best practices where **infrastructure is never exposed directly via raw endpoints (e.g., ALB DNS names)**.



View existing records
The following table lists the existing records in samuelonyejekwe.com.

5.2 Route 53 Hosted Zone Setup

Amazon Route 53 was configured as the **authoritative DNS service** for the domain.

Hosted Zone Configuration

- Type: **Public Hosted Zone**
- Domain: `citymart.samuelonyejekwe.com`

Role of Route 53

Route 53 performs **DNS resolution**, translating the domain name into the appropriate AWS resource endpoint.

Key Capabilities Leveraged

- Globally distributed DNS infrastructure
- Highly available and fault-tolerant resolution
- Native integration with AWS resources

From the implementation:

Route 53 is used as the authoritative DNS layer to route customer traffic to AWS resources such as ALB and CloudFront

Engineering Insight

Using Route 53 ensures:

- **Low-latency DNS resolution globally**
- Elimination of manual IP management
- Seamless integration with AWS-native services

5.3 Alias Record Configuration

Route 53 Alias records were used to map the domain to AWS resources.

Initial Configuration

Route 53 → Public ALB

Final Production Configuration

Route 53 → CloudFront → ALB

Alias Record Details

- Record Type: **A (Alias)**
- Target:
 - Initially: Public ALB
 - Final: CloudFront distribution

Create record [Info](#)

Quick create record [Switch to wizard](#)

▼ Record 1

Record name [Info](#)

.samuelonyejekwe.com

Keep blank to create a record for the root domain.

Record type [Info](#)

CNAME – Routes traffic to another domain name and to some AWS resources

☐ Alias

Value [Info](#)

Enter multiple values on separate lines.

TTL (seconds) [Info](#)

1m1h1d

TTL (seconds) [Info](#)

Recommended values: 60 to 172800 (two days)

Routing policy [Info](#)

Simple routing

Why Alias Records?

- No need for IP address management
- Direct integration with AWS services
- Automatic updates if underlying resources change

From the architecture evolution:

The system transitioned from direct ALB routing to CloudFront-based routing for improved performance and security

Engineering Benefit

Alias records enable:

- Cleaner architecture abstraction
- Reduced operational overhead
- Native AWS routing optimisation

5.4 SSL/TLS Implementation with ACM

AWS Certificate Manager (ACM) was used to provision and manage SSL/TLS certificates for secure communication.

Certificate Configuration

- Domains covered:
 - citymart.samuelonyejekwe.com
 - www.citymart.samuelonyejekwe.com
- Validation method: **DNS validation**

- Status: **Issued**



Integration Points

- Attached to:
 - CloudFront distribution (edge HTTPS)
 - Application Load Balancer (backend HTTPS support)

TLS Architecture

User → HTTPS → CloudFront

CloudFront → HTTP → ALB

Why This Design?

- Avoids certificate mismatch between CloudFront and ALB
- Simplifies certificate management
- Maintains secure edge communication

From our implementation:

TLS is terminated at CloudFront, while internal communication to the ALB remains HTTP for simplicity and compatibility

Security Benefits

- End-to-end encrypted client communication
- Automatic certificate renewal
- No manual key management

5.5 HTTPS Enforcement Strategy

The architecture enforces HTTPS at multiple layers to ensure **secure communication across all entry points**.

CloudFront Layer

- Viewer Protocol Policy:
 - Redirect HTTP → HTTPS

ALB Layer

- Listener configuration:
 - Port 80 → Redirect to HTTPS (301)
 - Port 443 → Forward to target group

From ALB configuration:

HTTP requests are automatically redirected to HTTPS, ensuring no plaintext traffic is served

Combined Enforcement Model

User → HTTP → Redirect → HTTPS → CloudFront → ALB

Engineering Outcome

- Eliminates insecure access paths
- Ensures compliance with modern security standards
- Improves SEO and browser trust

5.6 Evolution to Production Pattern (CloudFront Integration)

Initial Architecture

User → Route 53 → ALB → EC2

Final Production Architecture

User → Route 53 → CloudFront → ALB → EC2 → Backend → RDS

This transition represents a move from a **basic deployment model** to a **production-grade architecture**.

Why CloudFront Was Introduced

1. Performance Optimisation

- Edge caching reduces latency globally
- Static assets served from nearest location

2. Security Enhancement

- Acts as a secure entry point
- Provides AWS Shield (DDoS protection)
- Hides ALB from direct exposure

3. Scalability Improvement

- Offloads traffic from ALB and EC2
- Reduces backend load

4. Global Availability

- Distributed edge network ensures consistent performance worldwide

From our architecture:

CloudFront was introduced to provide caching, global delivery, and DDoS protection, forming the front layer of the system

The DNS and TLS layer of the CityMart architecture is designed as a **fully managed, secure, and globally optimised access layer**, combining:

- **Route 53** → DNS resolution
- **ACM** → automated TLS management
- **CloudFront** → global CDN and security edge
- **ALB** → application-level load balancing

Key Outcome

The system evolved from direct ALB exposure to a **Route 53 → CloudFront → ALB architecture**, which represents a **production-grade, secure, and scalable cloud design**.

6. Content Delivery Layer (CloudFront CDN)

6.1 CloudFront Architecture Integration

The CityMart architecture integrates Amazon CloudFront as a **global content delivery layer positioned at the edge of the system**, acting as the first point of contact for all incoming user traffic.

The final request flow is defined as:

User → CloudFront → Public ALB → Frontend EC2 → Internal ALB → Backend EC2 → RDS

This integration transforms the traditional 3-tier architecture into a **CDN-enhanced distributed system**, where CloudFront serves as:

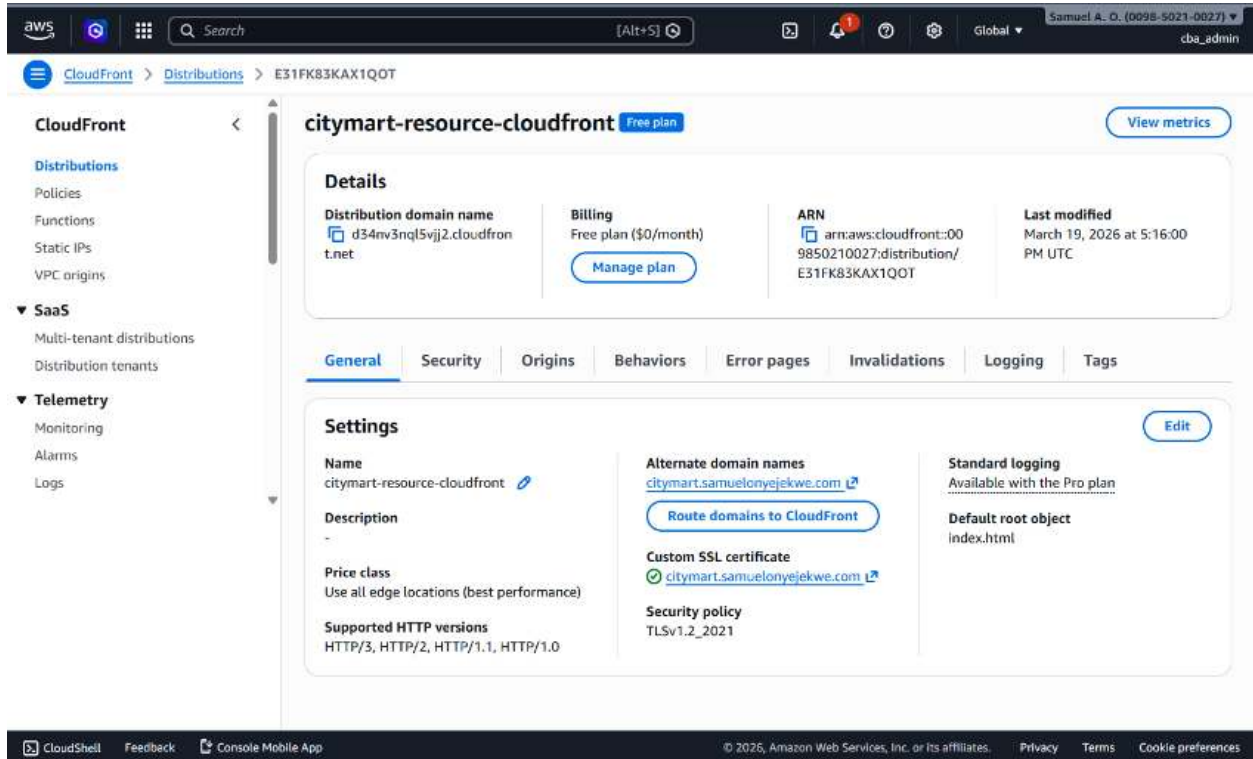
- A **global edge network**

- A content caching layer
- A secure entry point into the infrastructure

CloudFront sits in front of the public Application Load Balancer, ensuring that:

- Users access the system via either the custom domain or CloudFront distribution domain
- Requests are first processed at edge locations before reaching the origin infrastructure

This design introduces **geographical proximity between users and content delivery**, significantly improving response times while maintaining centralised backend processing.



6.2 Distribution Configuration

A CloudFront distribution was provisioned to serve as the CDN endpoint for the application.

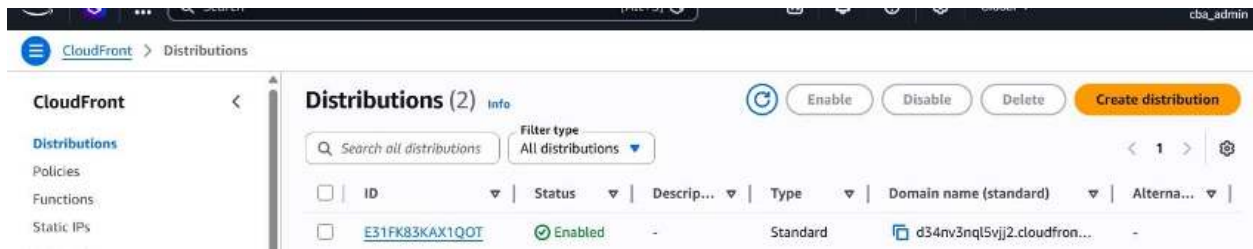
Key Distribution Properties

- **Distribution ID:** E31FK83KAX1QOT
- **Domain Name:** d34nv3nql5vjj2.cloudfront.net
- **Status:** Enabled

The distribution is configured to:

- Accept both HTTP and HTTPS requests
- Enforce HTTPS at the viewer level (via redirect policy)
- Support multiple HTTP methods required by the application:

- GET, HEAD, OPTIONS (static content)
- POST, PUT, PATCH, DELETE (API operations)



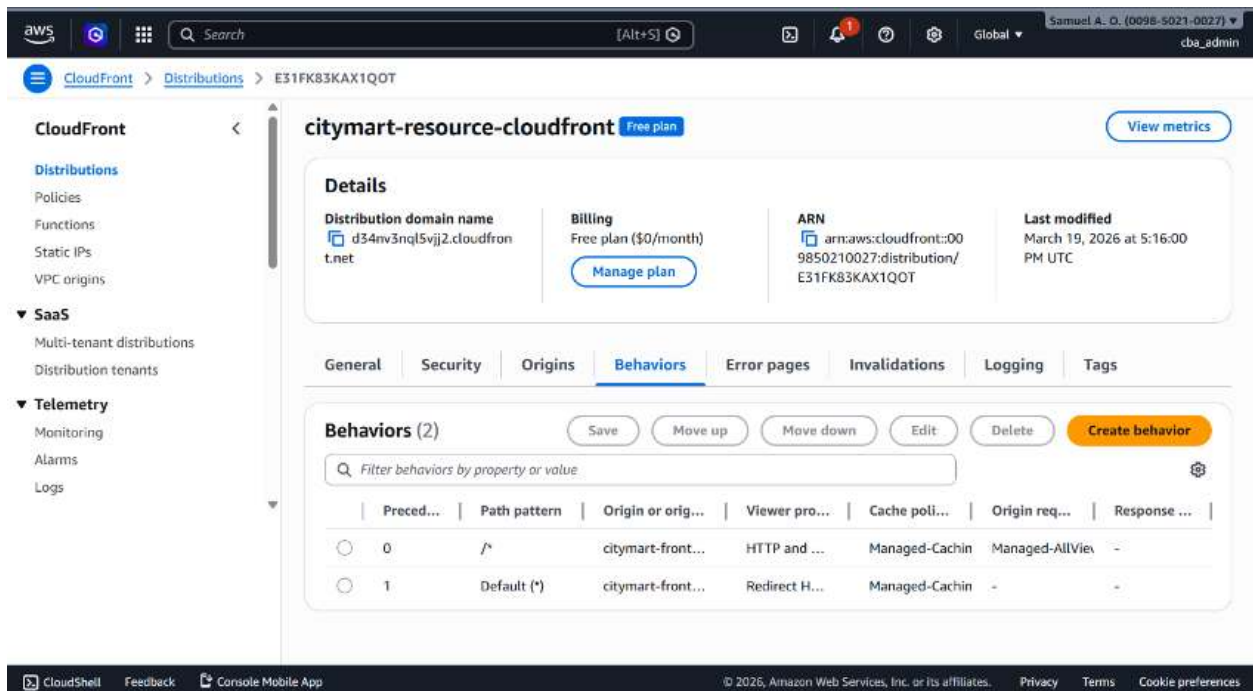
Two primary behaviours were configured:

Behaviour 1 - Application Traffic

- Path pattern: /*
- Origin: Frontend ALB
- Caching: Disabled (for dynamic content)
- Methods: Full HTTP method support

Behaviour 2 - Default Viewer Policy

- Redirect HTTP → HTTPS
- Optimised caching for general requests



This dual-behaviour configuration ensures that **dynamic application logic is preserved**, while still enforcing secure access and performance optimisation where applicable.

6.3 Origin Design (ALB as Origin)

The CloudFront distribution is configured to use the **Public Application Load Balancer as its origin**.

Origin Endpoint

citymart-frontend-alb-*.eu-west-2.elb.amazonaws.com

Rationale for ALB as Origin

Using the ALB as the origin enables:

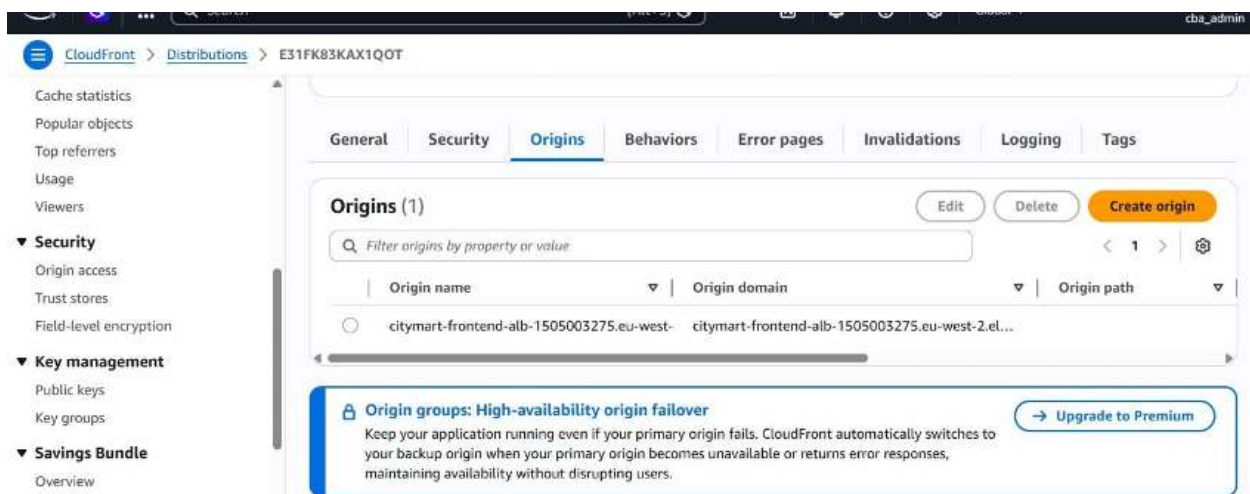
- **Load distribution across multiple frontend EC2 instances**
- **Integration with Auto Scaling Groups**
- **Health-aware traffic routing**

This design ensures that CloudFront does not communicate directly with EC2 instances, but instead relies on the ALB to:

- Route requests
- Perform health checks
- Maintain high availability

Protocol Design Decision

- **User → CloudFront: HTTPS**
- **CloudFront → ALB: HTTP**



This approach avoids SSL certificate hostname mismatch issues, since ALB certificates are issued for the application domain rather than the ALB DNS name.

This separation of concerns simplifies certificate management while maintaining secure client communication.

6.4 Caching Strategy (Static vs Dynamic Content)

A **selective caching strategy** was implemented to balance performance and data consistency.

Static Content (Cached)

CloudFront caches static assets such as:

- /assets/css/*
- /assets/js/*
- /assets/img/*

These assets are:

- Served from edge locations
- Cached for faster retrieval
- Offloaded from origin servers

Dynamic Content (Not Cached)

API endpoints such as:

- /api/products
- /api/product?id=1
- /api/order

are explicitly configured with:

CachingDisabled

Reasoning

Dynamic responses depend on real-time database state and must:

- Always reflect the latest data
- Avoid stale or inconsistent responses

This distinction ensures:

- **High performance for static delivery**
- **Data integrity for transactional operations**

This is a **production-grade caching pattern**, aligning with best practices for web applications.

6.5 Security Features (HTTPS, Shield, WAF Readiness)

CloudFront introduces a critical **security layer at the edge of the architecture**.

1. HTTPS Enforcement

- All client communication is encrypted via HTTPS
- HTTP requests are automatically redirected to HTTPS
- TLS certificates are managed through ACM

This ensures:

- Data confidentiality
- Protection against interception attacks

2. AWS Shield (DDoS Protection)

CloudFront provides built-in protection against distributed denial-of-service (DDoS) attacks:

- Absorbs malicious traffic at edge locations
- Prevents overload of origin infrastructure

3. Origin Protection

By placing CloudFront in front of the ALB:

- The ALB is no longer directly exposed to the internet
- Only CloudFront communicates with the origin

This reduces the attack surface and enforces a **controlled ingress point**

4. WAF Readiness

Although not implemented, the architecture is fully compatible with AWS WAF:

- Rules can be attached at the CloudFront layer
- Enables filtering of malicious requests (SQL injection, XSS, etc.)

6.6 Performance Optimisation (Edge Caching, Compression)

CloudFront significantly enhances system performance through multiple mechanisms.

1. Edge Caching

Content is cached at AWS edge locations worldwide:

- Users receive responses from the nearest location
- Reduces latency and round-trip time

2. Reduced Origin Load

By caching static content:

- Fewer requests reach the ALB and EC2 instances
- Backend resources are preserved for dynamic operations

3. Automatic Compression

CloudFront is configured with:

Compress objects automatically: Yes

Supported formats:

- Gzip
- Brotli

This results in:

- Smaller payload sizes
- Faster content delivery
- Reduced bandwidth consumption

4. Global Performance Consistency

Even though the infrastructure is deployed in a single region (eu-west-2), CloudFront ensures:

- Consistent performance globally
- Reduced latency for international users

6.7 Cache Invalidation Strategy

To maintain content freshness after updates, a **cache invalidation strategy** was implemented.

Invalidation Configuration

- Invalidation ID: I8XEB3I73HYBYKNKK8ZHK5R18M

- Path invalidated:

/*

Purpose

When new frontend content is deployed:

- Cached objects at edge locations may become outdated
- Invalidation forces CloudFront to fetch updated content from the origin

Outcome

- Ensures users receive the latest application version
- Prevents stale content delivery

This is particularly important for:

- UI updates
- JavaScript logic changes
- Asset modifications

The CloudFront layer in the CityMart architecture serves as a **critical edge component** that:

- Enhances **performance** through global caching and compression
- Improves **security** via HTTPS enforcement and DDoS protection
- Enables **scalability** by offloading origin traffic
- Maintains **data integrity** through selective caching strategies
- Supports **operational efficiency** via controlled invalidation

CloudFront was implemented as a globally distributed CDN layer to provide low-latency content delivery, secure edge termination, intelligent caching, and scalable traffic distribution in front of the application load balancer.

7. Application Load Balancers - ALBs

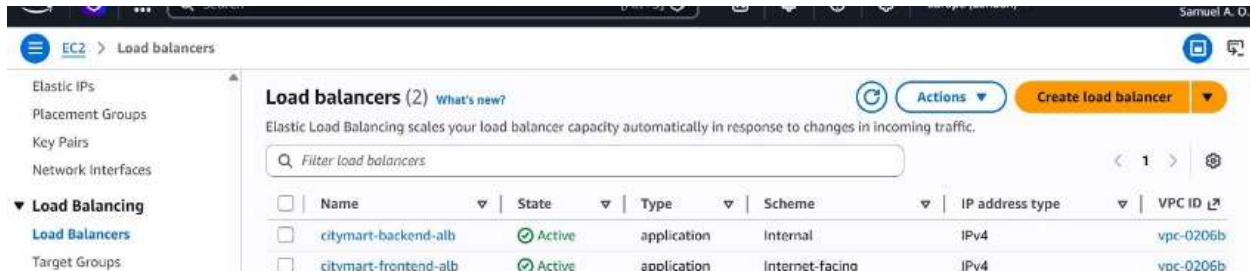
The CityMart platform implements a **dual-layer load balancing architecture** using Amazon Application Load Balancers (ALBs) to ensure **high availability, secure traffic distribution, and strict tier isolation** across the three-tier system.

This design consists of:

- **Frontend ALB (Internet-facing)** - entry point for user traffic
- **Backend ALB (Internal)** - private routing for application logic

Together, they enable a **controlled, scalable, and fault-tolerant request flow**:

User → CloudFront → Frontend ALB → Web Tier → Backend ALB → Application Tier → RDS



7.1 Frontend ALB (Internet-Facing)

The frontend ALB serves as the **public access layer** of the system and is deployed across **multiple Availability Zones** within the public subnets to ensure resilience and high availability.

HTTP → HTTPS Redirection

The ALB is configured with a listener on port **80 (HTTP)** that performs a **301 permanent redirect** to HTTPS (port 443).

- **Purpose:**
 - Enforces secure communication
 - Eliminates plaintext HTTP exposure
 - Improves SEO and browser caching behavior

From the implementation:

- Listener: HTTP : 80
- Action: **Redirect** → **HTTPS : 443**

Edit listener
Edit the protocol, port or default actions of your Application Load Balancer (ALB) listener.

► **Load balancer details:** citymart-frontend-alb

Listener details
A listener checks for connection requests using the protocol and port that you configure. The default action and any additional rules that you create determine how the Application Load Balancer routes requests to its registered targets.

Listener ARN
arn:aws:elasticloadbalancing:eu-west-2:009850210027:listener/app/citymart-frontend-alb/022583c2f9c0ad02/d26d7f1f6a195b83

Protocol
Used for connections from clients to the load balancer.
HTTP

Port
The port on which the load balancer is listening for connections.
80
1-65535

Default action [Info](#)
The default action is used if no other rules apply. Choose the default action for traffic on this listener.

Routing action

☐ Forward to target groups ☒ **Redirect to URL** ☐ Return fixed response

Redirect to URL [Info](#)
Redirect client requests from one URL to another. You cannot redirect HTTPS to HTTP. To avoid a redirect loop, you must modify at least one of the following components: protocol, port, hostname or path. Components that you do not modify retain their original values.

URI parts **Full URL**

Protocol
Used for connections from clients to the load balancer.
HTTPS

Port
The port on which the load balancer is listening for connections.
443
1-65535 or to retain the original port enter #{port}

☐ Custom host, path, query
Select to modify host, path and query. If no changes are made, settings from the request URL are retained.

Status code
301 - Permanently moved

This ensures that **all client traffic is encrypted before entering the system**, aligning with modern security best practices

HTTPS Listener with ACM

The HTTPS listener on port **443** is configured with an **AWS Certificate Manager (ACM)** certificate.

- TLS policy: modern (TLS 1.2 / 1.3)
- Certificate: issued via ACM with DNS validation

Key benefits:

- End-to-end encryption (user → ALB)
- Automated certificate renewal
- Native AWS integration with Route 53

Traffic flow at this stage:

User → HTTPS → Frontend ALB (TLS terminated here)

This design ensures that **encryption is handled at the edge of the application layer**, simplifying backend communication while maintaining strong security guarantees.

Target Group Integration

The frontend ALB forwards traffic to a **target group containing frontend EC2 instances (Nginx servers)**.

Flow:

Frontend ALB → Target Group → Frontend EC2 Instances

The target group acts as a **logical bridge** between the load balancer and compute layer:

- Routes traffic to multiple instances
- Distributes load evenly
- Enables dynamic scaling via ASG

From our architecture:

- Target Group: `citymart-frontend-alb-tg`
- Targets: EC2 instances in Auto Scaling Group

This integration allows:

- Automatic registration of new instances
- Seamless removal of unhealthy instances
- Continuous service availability

7.2 Backend ALB (Internal)

The backend ALB is deployed as an **internal (private) load balancer**, operating entirely within private subnets.

Private Routing

Unlike the frontend ALB, the backend ALB:

- Has **no public IP**
- Is **not accessible from the internet**
- Only accepts traffic from the **frontend EC2 security group**

Flow:

Frontend EC2 → Internal ALB → Backend EC2

This ensures that:

- Backend services are **completely hidden from public access**
- All API traffic is routed internally

Application Isolation

The backend ALB enforces strict **tier separation**:

- Only frontend servers can call backend services
- Backend servers are isolated from direct user interaction

Security model:

Layer	Accessible By
Frontend ALB	Internet
Frontend EC2	Frontend ALB only
Backend ALB	Frontend EC2 only
Backend EC2	Backend ALB only

This architecture achieves:

- **Zero-trust layered design**
- Reduced attack surface
- Controlled inter-service communication

7.3 Target Groups and Health Checks

Target groups are central to how ALBs distribute traffic and maintain system reliability.

Target Groups per Tier

Two target groups are defined:

- **Frontend Target Group**
 - Targets: Nginx EC2 instances
- **Backend Target Group**
 - Targets: PHP API EC2 instances

Each ALB forwards traffic to its respective target group.

Load Balancing Function

The ALB distributes incoming requests using standard algorithms such as:

- Round-robin
- Least outstanding requests

Example:

Request 1 → EC2-A

Request 2 → EC2-B

Request 3 → EC2-A

This ensures:

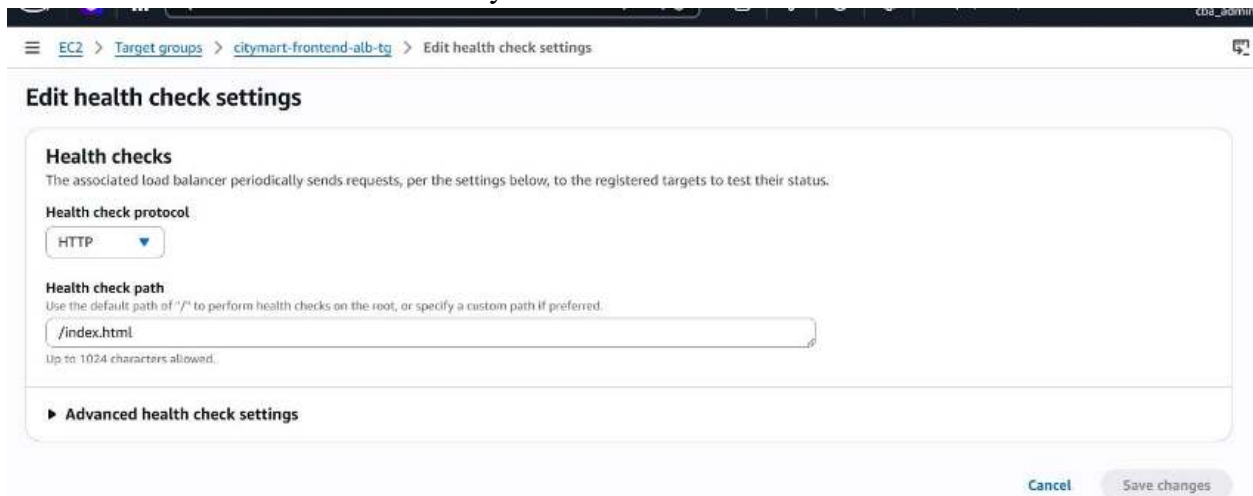
- Even load distribution
- Optimal resource utilisation

Health Checks

Health checks are configured to ensure traffic is only sent to **healthy instances**.

Frontend Health Check

- Endpoint: /index.html
- Verifies web server availability



The screenshot shows the AWS Management Console interface for editing health check settings. The breadcrumb navigation at the top reads: EC2 > Target groups > citymart-frontend-alb-tg > Edit health check settings. The main heading is 'Edit health check settings'. Below this, there is a section titled 'Health checks' with a description: 'The associated load balancer periodically sends requests, per the settings below, to the registered targets to test their status.' Under 'Health check protocol', a dropdown menu is set to 'HTTP'. Under 'Health check path', a text input field contains '/index.html' with a note below it stating 'Up to 1024 characters allowed.' At the bottom of the settings panel, there is a link 'Advanced health check settings'. To the right of the settings panel are two buttons: 'Cancel' and 'Save changes'.

Backend Health Check

- Endpoint: /api/products (real API endpoint)
- Verifies application functionality (not just server uptime)

The screenshot shows the 'Edit health check settings' page in the AWS Management Console. The breadcrumb navigation at the top reads: EC2 > Target groups > citymart-backend-alb-tg > Edit health check settings. The main content area is titled 'Edit health check settings' and contains a section for 'Health checks'. Below this, there is a description: 'The associated load balancer periodically sends requests, per the settings below, to the registered targets to test their status.' The 'Health check protocol' is set to 'HTTP' via a dropdown menu. The 'Health check path' is set to '/api/products' in a text input field, with a note below it stating 'Use the default path of "/" to perform health checks on the root, or specify a custom path if preferred.' and 'Up to 1024 characters allowed.' At the bottom of the settings section is a link '► Advanced health check settings'. At the bottom right of the page are 'Cancel' and 'Save changes' buttons.

If an instance fails:

- It is marked **unhealthy**
- ALB **stops routing traffic** to it

This enables:

- Fault detection
- Automatic traffic rerouting
- Improved reliability

Integration with Auto Scaling

Target groups are tightly integrated with Auto Scaling Groups:

- New EC2 instance launched → automatically registered
- Instance terminated → automatically deregistered

This creates a **self-healing system**:

Instance failure → ASG launches new instance → joins target group → receives traffic

7.4 Request Routing Logic

The request routing across the system follows a **layered, controlled path**, ensuring both performance and security.

End-to-End Flow

1. User Request

User → Route53 → CloudFront

2. Entry Layer

CloudFront → Frontend ALB (HTTPS)

3. Web Tier Routing

Frontend ALB → Target Group → Nginx EC2

4. Application Routing

Nginx (/api/*) → Internal ALB

5. Backend Processing

Internal ALB → Backend Target Group → PHP EC2

6. Data Access

Backend EC2 → RDS MySQL

Layered Routing Logic

Layer	Function
CloudFront	Edge routing + caching
Frontend ALB	Public traffic distribution
Nginx	Static serving + API proxy
Backend ALB	Internal API routing
Backend EC2	Business logic
RDS	Data persistence

Key Architectural Benefits

1. High Availability

- Multi-AZ ALB deployment
- Health checks + failover

2. Security

- Internal backend isolation
- HTTPS enforced at entry

3. Scalability

- ASG + target group integration
- Dynamic traffic distribution

4. Operational Efficiency

- Clear separation of responsibilities
- Easy debugging and monitoring

The ALB architecture in CityMart demonstrates a **production-grade, layered load balancing strategy** where:

- The **frontend ALB** handles secure internet traffic and distributes it across web servers
- The **backend ALB** ensures private, controlled routing of application requests
- **Target groups + health checks** provide resilience and automatic failure handling
- **Routing logic enforces strict tier isolation**, aligning with AWS Well-Architected principles

This design achieves **secure, scalable, fault-tolerant request handling across all tiers of the application**.

8. Compute Layer (EC2, AMI, Launch Templates, ASG)

The compute layer of the CityMart architecture is designed as a **fully elastic, highly available, and self-healing system**, leveraging Amazon EC2 instances orchestrated through Auto Scaling Groups and Launch Templates. This layer forms the execution backbone of both the **presentation (frontend)** and **application (backend)** tiers.

The architecture follows a **stateless, horizontally scalable model**, ensuring that compute resources can dynamically adjust to workload demands while maintaining operational consistency and resilience.

8.1 EC2 Instance Configuration

The CityMart platform utilises Amazon EC2 instances as the primary compute resource for both frontend and backend tiers. These instances are deliberately deployed within **private subnets**, ensuring that they are not directly exposed to the public internet.

Network Info

For most applications, you can use multiple Availability Zones and let EC2 Auto Scaling balance your instances across the zones. The default VPC and default subnets are suitable for getting started quickly.

VPC

Choose the VPC that defines the virtual network for your Auto Scaling group.

vpc-0206bba22a724b50b (citymart-3-tier-architecture-vpc)
10.0.0.0/16



[Create a VPC](#)

Availability Zones and subnets

Define which Availability Zones and subnets your Auto Scaling group can use in the chosen VPC.

Select Availability Zones and subnets



euw2-az2 (eu-west-2a) | subnet-0056346b219601a91 (web-private-subnet-2a)
10.0.48.0/20

euw2-az3 (eu-west-2b) | subnet-0c2983171f07ce599 (web-private-subnet-2b)
10.0.64.0/20

[Create a subnet](#)

Availability Zone distribution - new

Auto Scaling automatically balances instances across Availability Zones. If launch failures occur in a zone, select a strategy.

☒ **Balanced best effort**
If launches fail in one Availability Zone, Auto Scaling will attempt to launch in another healthy Availability Zone.

☐ **Balanced only**
If launches fail in one Availability Zone, Auto Scaling will continue to attempt to launch in the unhealthy Availability Zone to preserve balanced distribution.

VPC

Choose the VPC that defines the virtual network for your Auto Scaling group.

vpc-0206bba22a724b50b (citymart-3-tier-architecture...
10.0.0.0/16



[Create a VPC](#)

Availability Zones and subnets

Define which Availability Zones and subnets your Auto Scaling group can use in the chosen VPC.

Select Availability Zones and subnets



euw2-az2 (eu-west-2a) | subnet-0c8a3c4c0ae7b2d7a (app-private-subnet-3a)
10.0.96.0/20

euw2-az3 (eu-west-2b) | subnet-05e844ab0f8a66002 (app-private-subnet-3b)
10.0.112.0/20

[Create a subnet](#)

Frontend EC2 (Web Tier)

- Runs **Nginx web server**
- Hosts static frontend assets:

- HTML pages (index, products, checkout)
 - CSS, JavaScript, images
- Acts as a **reverse proxy**, forwarding /api requests to the backend via the internal load balancer

Backend EC2 (Application Tier)

- Runs **Apache + PHP**
- Hosts API endpoints:
 - /api/products
 - /api/product
 - /api/order
- Executes business logic and communicates with the RDS database

Key Configuration Characteristics

- Instance type: **t3.micro** (cost-efficient, burstable performance)
- Instances deployed across **multiple Availability Zones**
- No public IP assignment (private-only architecture)
- Access managed via **AWS Systems Manager (SSM)** instead of SSH

Engineering Insight

This configuration ensures:

- Reduced attack surface (no direct internet exposure)
- Controlled access via ALBs and SSM
- Clear separation of concerns between tiers

8.2 Amazon Machine Images (AMI)

Amazon Machine Images (AMIs) are used to standardise the compute environment across all EC2 instances.

Each instance is launched from a predefined AMI that contains:

- Operating system (Amazon Linux)
- Required runtime environment (Nginx or Apache + PHP)
- Preconfigured dependencies and system packages

Role of AMIs in the Architecture

- Ensure **consistent environment replication**
- Enable rapid provisioning of new instances
- Support **immutable infrastructure practices**

From our implementation:

- AMIs are referenced within launch templates
- Instances inherit configuration automatically during launch

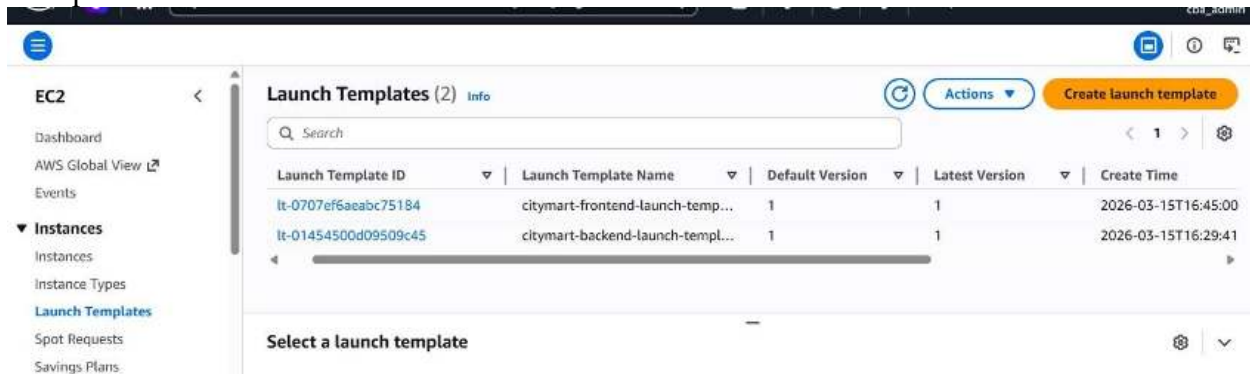
Engineering Insight

Using AMIs eliminates configuration drift and guarantees that every instance launched:

- behaves identically
- integrates seamlessly with load balancers
- passes health checks consistently

8.3 Launch Templates Design

Launch Templates define the blueprint for provisioning EC2 instances within the Auto Scaling Groups.



Each tier has a **dedicated launch template**:

Frontend Launch Template

- AMI with Nginx environment
- Instance type (t3.micro)
- Security group: frontend server SG
- User data script:
 - Pulls frontend code from S3
 - Configures Nginx
 - Starts web service

Amazon Machine Image (AMI)

citymart-frontend-ami
ami-0815e58adf0565fe4
2026-03-15T15:33:23.000Z Virtualization: hvm ENA enabled: true
Root device type: ebs Boot mode: uefi-preferred

Description
Web-tier-resources

Architecture	AMI ID
x86_64	ami-0815e58adf0565fe4

▼ **Instance type** [Info](#) | [Get advice](#) Advanced

Instance type

t3.micro
Family: t3 2 vCPU 1 GiB Memory Current generation: true
On-Demand Linux base pricing: 0.0118 USD per Hour
On-Demand SUSE base pricing: 0.0118 USD per Hour
On-Demand Ubuntu Pro base pricing: 0.0153 USD per Hour
On-Demand RHEL base pricing: 0.0406 USD per Hour

☒ All generations [Compare instance types](#)

On-Demand Windows base pricing: 0.021 USD per Hour

Additional costs apply for AMIs with pre-installed software

```

# Health Checks
# -----

echo "==== Nginx Service Status ====="
sudo systemctl is-active nginx

echo "==== Local HTTP Test ====="
curl -I http://localhost

echo "==== Backend API Test Through Proxy ====="
curl -s http://localhost/api/health || true

echo "==== Frontend Initialization Complete ====="

```

☐ User data has already been base64 encoded

Backend Launch Template

- AMI with Apache + PHP
- Instance type (t3.micro)
- Security group: backend server SG
- User data script:
 - Pulls backend code from S3
 - Applies database configuration
 - Starts API service

Key Capabilities

- Defines:
 - AMI
 - Instance type
 - IAM role
 - Security groups
 - User data scripts
- Ensures **repeatable, version-controlled infrastructure**

Amazon Machine Image (AMI)

citymart-backend-ami
ami-00a821d69e6af7b87
2026-03-15T15:29:34.000Z Virtualization: hvm ENA enabled: true
Root device type: ebs Boot mode: uefi-preferred

Description
App-tier-resources

Architecture	AMI ID
x86_64	ami-00a821d69e6af7b87

▼ Instance type [Info](#) | [Get advice](#) Advanced

Instance type

t3.micro
Family: t3 2 vCPU 1 GiB Memory Current generation: true
On-Demand Linux base pricing: 0.0118 USD per Hour
On-Demand SUSE base pricing: 0.0118 USD per Hour
On-Demand Ubuntu Pro base pricing: 0.0153 USD per Hour
On-Demand RHEL base pricing: 0.0406 USD per Hour
On-Demand Windows base pricing: 0.021 USD per Hour

☒ All generations [Compare instance types](#)

Additional costs apply for AMIs with pre-installed software

User data - optional [Info](#)

Upload a file with your user data or enter it in the field.

[Choose file](#)

```
echo "==== API Health Check ====="
curl -s http://localhost/api/health.php || true

echo "==== Products API Test ====="
curl -s http://localhost/api/products.php | head -n 200 || true

echo "==== Database Connectivity Test ====="
mysql -h citymart-database.cdgowmasatge.eu-west-2.rds.amazonaws.com \
-u admin \
-p phoenix80384citymart \
-D citymart \
-e "SHOW TABLES;"

echo "==== Backend Initialization Complete ====="
```

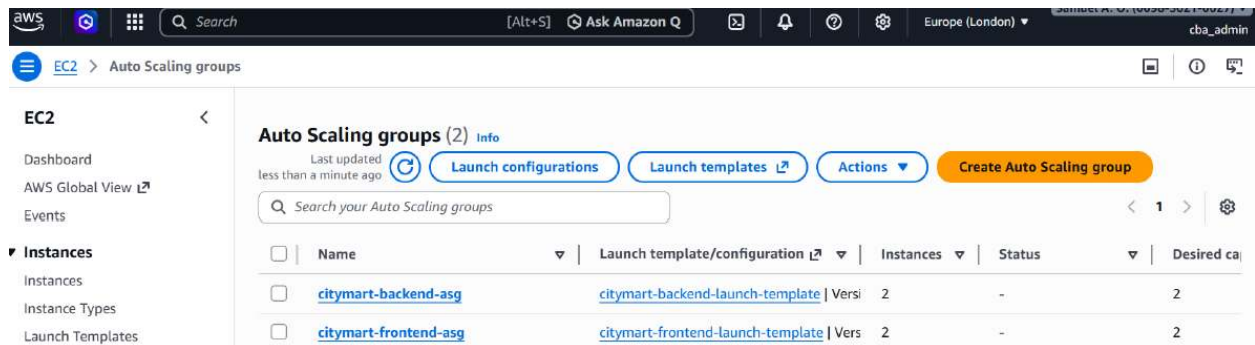
Engineering Insight

Launch templates enable:

- **Immutable deployments**
- Rapid scaling without manual configuration
- Consistency across all instances in the system

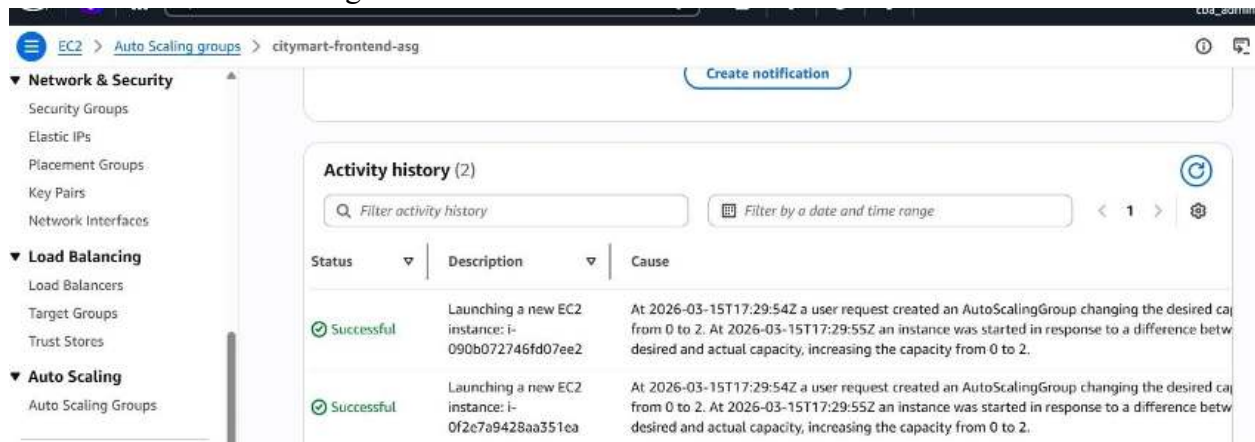
8.4 Auto Scaling Groups (Frontend & Backend)

The compute layer is orchestrated using two independent Auto Scaling Groups (ASGs):



Frontend ASG

- Manages Nginx web servers
- Connected to **public ALB target group**
- Handles user-facing traffic



The screenshot shows the AWS Management Console for the 'citymart-frontend-asg' Auto Scaling group. The 'Instance management' tab is active, displaying a table of instances. The table has columns for Instance ID, Lifecycle, Instance type, Weight, Launch name, Availability zone, Health status, and Protection state. Two instances are listed, both in 'InService' lifecycle and 'Healthy' health status. Below the table, there is a section for 'Instance lifecycle policy for lifecycle hooks' with a 'Manage policy' button.

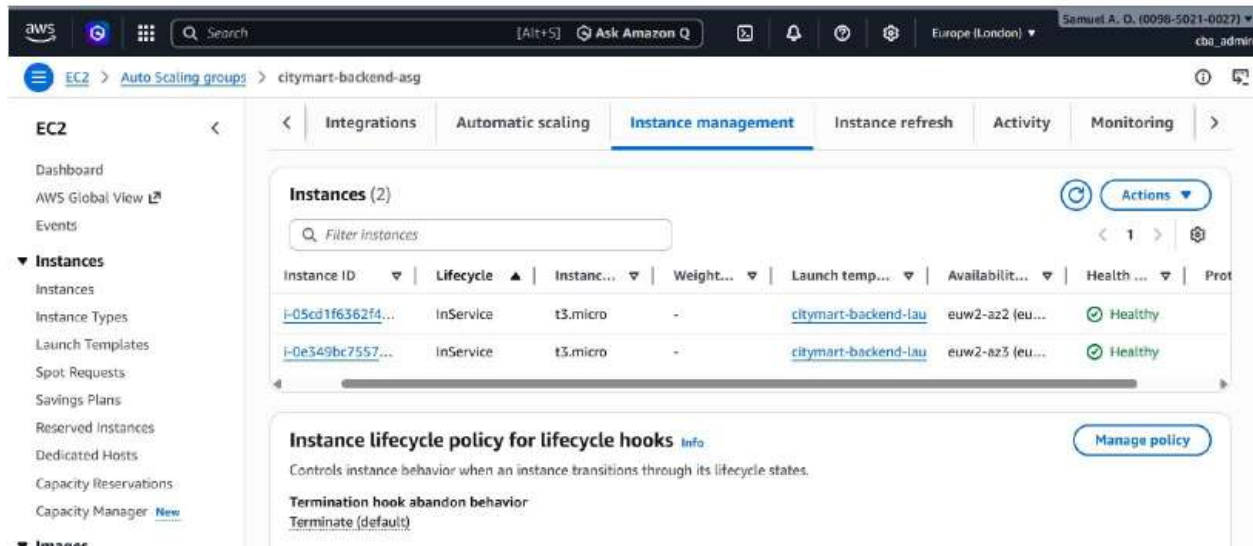
Instance ID	Lifecycle	Instance type	Weight	Launch name	Availability zone	Health status	Protection state
i-090b072746fd0...	InService	t3.micro	-	citymart-frontend	euw2-az2	Healthy	Protected
i-0f2e7a9428aa3...	InService	t3.micro	-	citymart-frontend	euw2-az3	Healthy	Protected

Backend ASG

- Manages application servers
- Connected to **internal ALB target group**
- Processes API requests

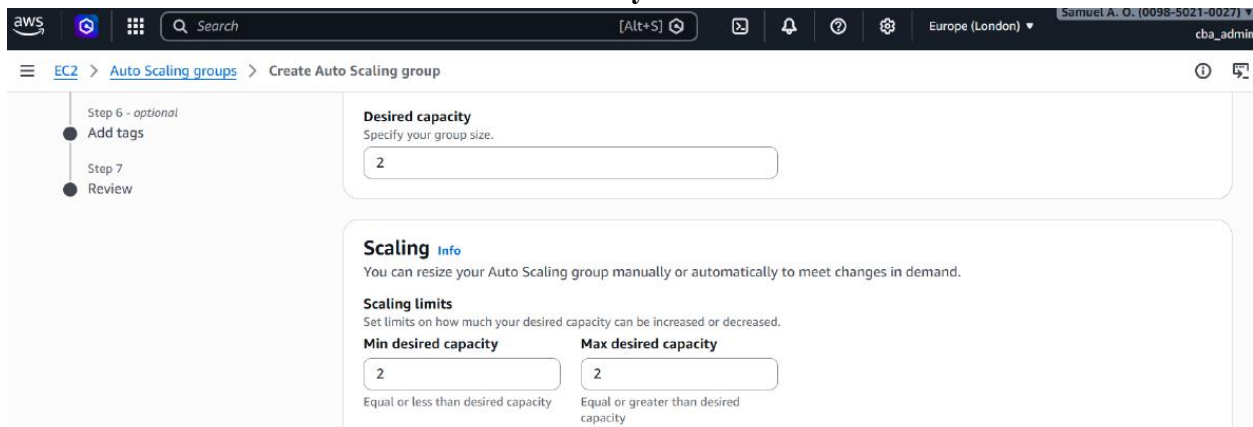
The screenshot shows the AWS Management Console for the 'citymart-backend-asg' Auto Scaling group. The 'Activity history' tab is active, displaying a table of activities. The table has columns for Status, Description, and Cause. Two activities are listed, both with a 'Successful' status. The activities describe the launching of new EC2 instances in response to a change in the desired capacity of the Auto Scaling group.

Status	Description	Cause
Successful	Launching a new EC2 instance: i-0e349bc755723515f	At 2026-03-15T17:59:03Z a user request created an AutoScalingGroup changing the desired capacity from 0 to 2. At 2026-03-15T17:59:06Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 0 to 2.
Successful	Launching a new EC2 instance: i-05cd1f6362f48b60a	At 2026-03-15T17:59:03Z a user request created an AutoScalingGroup changing the desired capacity from 0 to 2. At 2026-03-15T17:59:06Z an instance was started in response to a difference between desired and actual capacity, increasing the capacity from 0 to 2.



Configuration

- Minimum capacity: **2 instances**
- Desired capacity: **2 instances**
- Instances distributed across **2 Availability Zones**



Integration Flow

Auto Scaling Group
 ↓
 Registers instances into
 ↓
 Target Group
 ↓
 Used by Load Balancer

Key Capabilities

- Automatic instance launch and termination
- Health-based replacement

- Dynamic scaling based on demand

Engineering Insight

Separate ASGs for frontend and backend allow:

- Independent scaling of tiers
- Better resource utilisation
- Isolation of performance bottlenecks

8.5 Scaling Behaviour and Self-Healing

The CityMart architecture implements **dynamic scaling and automatic recovery mechanisms** to ensure system resilience.

Scaling Behaviour

- ASGs adjust capacity based on demand
- Instances can scale from:
 - **0** → **active capacity**
 - Automatically provisioned when needed

Self-Healing Mechanism

When an instance fails or is terminated:

1. Detection

- ASG detects mismatch:
 - $\text{Actual capacity} < \text{Desired capacity}$

2. Replacement

- New instance launched automatically using launch template

3. Registration

- Instance added to target group
- Must pass health checks before receiving traffic

4. Traffic Restoration

- Load balancer resumes normal routing

Load Balancer Interaction

- Unhealthy instances:
 - Removed from target group
- Healthy instances:
 - Receive traffic

Engineering Insight

This creates a **fully autonomous recovery system**:

- No manual intervention required
- Zero downtime during instance failure
- Continuous service availability

8.6 Multi-AZ Deployment Strategy

High availability is achieved through **multi-Availability Zone deployment**.

Implementation

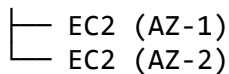
- Instances deployed across:
 - eu-west-2a
 - eu-west-2b
- Load balancers span both AZs
- ASGs distribute instances evenly

Benefits

- Fault tolerance:
 - If one AZ fails → traffic shifts automatically
- High availability:
 - Continuous operation under failure scenarios
- Resilience:
 - No single point of failure

Traffic Distribution

ALB



Engineering Insight

Multi-AZ design ensures:

- Infrastructure-level redundancy
- Application-level resilience
- Alignment with AWS Well-Architected reliability principles

The compute layer of the CityMart architecture demonstrates a **production-grade, cloud-native design** characterised by:

- **Stateless EC2 instances** deployed in private subnets
- **Launch templates** enabling consistent and repeatable provisioning
- **Auto Scaling Groups** providing elasticity and self-healing
- **Multi-AZ deployment** ensuring high availability and fault tolerance
- Tight integration with **load balancers and target groups** for traffic distribution

Overall, this layer achieves:

- **Elastic scalability**
- **Operational automation**
- **High resilience**
- **Security through isolation**

9. Application Architecture

The CityMart application follows a structured **three-tier application architecture**, where responsibilities are clearly separated between the presentation layer (frontend), application layer (backend), and data layer (database). This separation ensures scalability, maintainability, and clean request handling across the system.

The application is deployed using a **decoupled codebase stored in Amazon S3**, with each tier independently provisioned and configured during instance initialisation.

9.1 Frontend Application Structure

The frontend application represents the **presentation layer** of the system and is responsible for delivering the user interface and interacting with backend APIs.

Structure and Components

The frontend code is stored in the S3 bucket under:

`citymart-frontend/`

This directory contains:

- HTML pages:
 - `index.html`
 - `products.html`
 - `product.html`
 - `checkout.html`
 - `order-success.html`

- Static assets:
 - CSS (styles.css)
 - JavaScript (api.js, products.js, product.js, checkout.js)
 - Images (rice.png, oil.png, soap.png, etc.)

These files are deployed to the EC2 instances under:

`/var/www/html`

and served by the Nginx web server.

Frontend Responsibilities

The frontend layer performs the following functions:

- Renders user-facing pages for product browsing and ordering
- Handles user interaction through JavaScript
- Sends HTTP requests to backend APIs
- Displays dynamic data retrieved from the backend

Example API interactions from the frontend:

```
GET /products
GET /product?id=1
POST /order
```

Static Asset Delivery

Static resources such as images, CSS, and JavaScript files are:

- Served directly by Nginx
- Cached using HTTP headers (Cache-Control, expiry settings)
- Delivered efficiently to users

Additionally, when integrated with CloudFront:

- These assets are cached at edge locations
- Latency is reduced for global users
- Backend load is significantly reduced

9.2 Backend Application Structure

The backend application represents the **business logic layer** and is responsible for processing requests, executing application logic, and interacting with the database.

Structure and Components

The backend code is stored in:

`citymart-backend/`

Key components include:

- API endpoints:
 - `products.php`
 - `product.php`
 - `order.php`
 - `health.php`
- Configuration files:
 - `app.php`
 - `database.php`
- Core services:
 - `CustomersService.php`
 - `OrdersService.php`
 - Database abstraction classes
- Scripts:
 - `migrate.sql`
 - `seed.sql`
 - `reset_db.sh`
- Entry point:
 - `public/index.php`

This structure enforces a **modular backend design**, separating:

- routing
- configuration
- business logic
- database operations

Backend Responsibilities

The backend layer is responsible for:

- Processing API requests from the frontend
- Executing business logic (product retrieval, order handling)
- Performing database operations
- Returning structured JSON responses

Example response:

```
{
  "ok": true,
  "data": {
    "products": [...]
  }
}
```

9.3 API Design and Endpoints

The CityMart backend exposes a set of REST-style API endpoints designed for simplicity and clarity.

Core Endpoints

1. */products*

- Method: GET
- Function: Retrieve all available products
- Used by: `products.html`

2. */product*

- Method: GET
- Parameters: `id`
- Function: Retrieve a single product's details

3. */order*

- Method: POST
- Function: Submit a customer order

API Design Characteristics

- Stateless design (no session storage on server)
- JSON-based responses
- Clear separation of endpoints
- Simple REST-like structure

These APIs are accessed via:

`/api/*`

and routed internally to backend services via the Nginx reverse proxy.

9.4 Application Request Flow

The application follows a clearly defined **end-to-end request lifecycle**:

Step-by-Step Flow

1. User accesses application via browser
2. Request is routed through CloudFront and ALB
3. Request reaches frontend EC2 instance (Nginx)
4. Static content is served directly (HTML, CSS, JS)
5. API request is triggered (e.g., /api/products)
6. Nginx forwards request to internal ALB
7. Internal ALB routes request to backend EC2
8. Backend processes logic and queries database
9. Response is returned to frontend
10. Frontend renders data to user

Full flow confirmed in architecture:

User → CloudFront → ALB → Frontend → Backend → RDS

9.5 Nginx Reverse Proxy Configuration

Nginx serves as the **frontend web server and API gateway** within the web tier.

Key Responsibilities

- Serve static frontend content
- Route API requests to backend
- Apply caching headers
- Provide health check endpoint

Core Configuration Behaviour

Static Content Serving

```
root /usr/share/nginx/html;
```

Health Check Endpoint

```
location = /health {  
    return 200 "OK";  
}
```

Static Asset Caching

```
location /assets/ {  
    expires 7d;  
    add_header Cache-Control "public, max-age=604800";  
}
```

Reverse Proxy for API

```
location /api/ {  
    proxy_pass http://INTERNAL_ALB_DNS;  
}
```

This configuration ensures:

- All /api/* requests are forwarded to backend
- Backend remains isolated from public access
- Clean separation between UI and logic layers

Nginx acts as a controlled gateway between frontend and backend

9.6 Apache Backend Configuration

The backend EC2 instances run Apache to serve PHP-based APIs.

Core Configuration

- Apache listens on port 80
- Backend is hosted internally behind ALB
- API endpoints exposed under /api/*

Key Configuration Elements

Document Root

/var/www/html

API Routing

Alias /api /var/www/html/api

Directory Permissions

Require all granted

Security and Headers

X-Content-Type-Options

X-Frame-Options

Referrer-Policy

These headers enhance security even within internal communication layers.

Design Considerations

- Backend is not publicly exposed
- Only accessible via internal ALB
- Relies on forwarded headers from Nginx

TLS Handling

- HTTPS terminates at Public ALB
- Backend communicates over HTTP internally
- Secure communication maintained via controlled network boundaries

Backend operates fully within private network boundaries

The CityMart application architecture demonstrates:

- Clear **separation of concerns** between frontend and backend
- Stateless, scalable API design
- Secure internal communication via reverse proxy and internal ALB
- Efficient static content delivery and caching strategy
- Production-grade deployment using S3, ASG, and private networking

10. Application Deployment Model

The CityMart platform implements a **fully automated, S3-driven deployment model** designed to support scalability, consistency, and operational efficiency across both the frontend and backend tiers.

This deployment approach eliminates manual configuration on instances and ensures that all compute resources remain **stateless, reproducible, and aligned with infrastructure-as-code principles**.

The deployment model is tightly integrated with:

- **Amazon S3** for centralised code storage
- **EC2 Auto Scaling Groups (ASGs)** for compute provisioning
- **Launch Templates and user data scripts** for bootstrapping
- **AWS Systems Manager (SSM)** for controlled remote execution

This architecture enables **zero-touch deployments**, rapid updates, and seamless scaling across multiple Availability Zones.

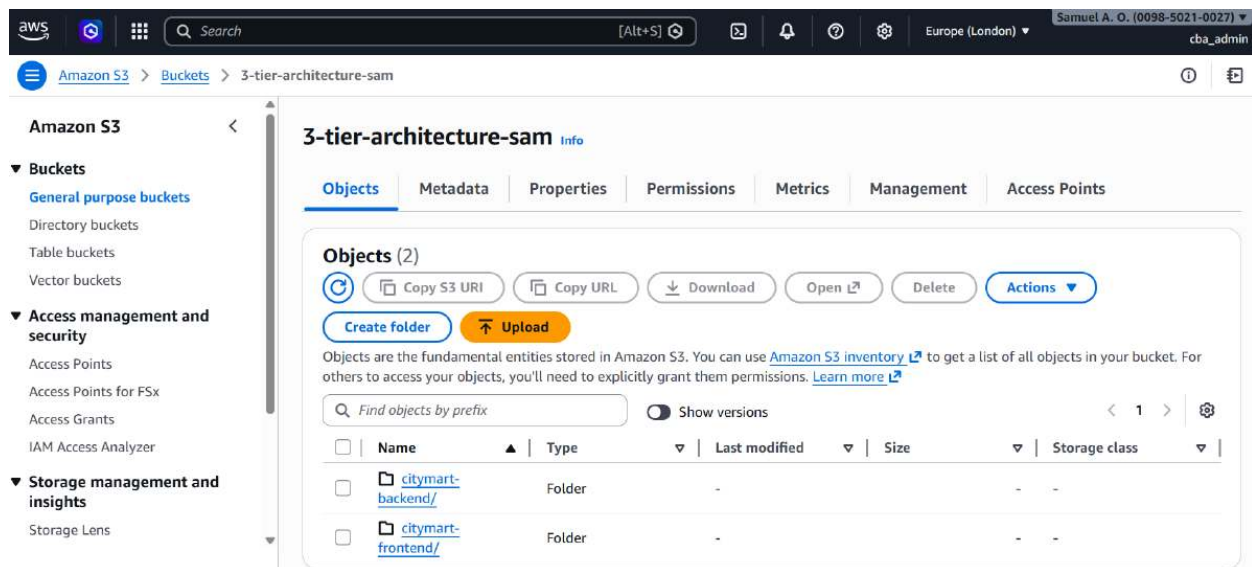
10.1 S3-Based Code Storage

The CityMart application code is centrally stored in an Amazon S3 bucket:

3-tier-architecture-sam

Within this bucket, the application is logically separated into two distinct folders:

citymart-frontend/
citymart-backend/



As defined in our implementation

Upload succeeded
For more information, see the Files and folders table.

Upload: status Close

After you navigate away from this page, the following information is no longer available.

Summary

Destination	Succeeded	Failed
s3://3-tier-architecture-sam	38 files, 63.0 KB (100.00%)	0 files, 0 B (0%)

Files and folders | Configuration

Files and folders (38 total, 63.0 KB)

Find by name

Name	Folder	Type	Size	Status	Error
Validator.php	application-code/backe...	-	1.5 KB	Succeeded	-
Response.php	application-code/backe...	-	4.0 KB	Succeeded	-
seed.sql	application-code/backe...	-	491.0 B	Succeeded	-
migrate.sql	application-code/backe...	-	1.7 KB	Succeeded	-
reset_db.sh	application-code/backe...	text/x-sh	599.0 B	Succeeded	-
products.js	application-code/front...	text/javascript	2.4 KB	Succeeded	-
config.js	application-code/front...	text/javascript	865.0 B	Succeeded	-
api.js	application-code/front...	text/javascript	930.0 B	Succeeded	-
product.js	application-code/front...	text/javascript	1.8 KB	Succeeded	-
checkout.js	application-code/front...	text/javascript	2.1 KB	Succeeded	-

Design Rationale

This separation aligns directly with the 3-tier architecture:

Tier	S3 Folder	Purpose
Frontend	citymart-frontend	Static UI (HTML, CSS, JS, images)
Backend	citymart-backend	API logic (PHP, configs, services)

Key Characteristics

- **Decoupled deployment:** Each tier pulls only the code it requires
- **Centralised version control point**
- **No code baked into AMIs** (supports immutability)
- **Supports horizontal scaling seamlessly**

Each Auto Scaling Group retrieves its respective code during instance initialisation, ensuring that **every instance is identically configured at launch**.

10.2 Deployment Strategy (S3 Sync)

The deployment mechanism is based on **runtime code retrieval using aws s3 sync**, executed during instance bootstrapping or via SSM commands.

Frontend Deployment Example

```
aws s3 sync s3://3-tier-architecture-sam/citymart-frontend /var/www/html
```

Backend Deployment Example

```
aws s3 sync s3://3-tier-architecture-sam/citymart-backend /var/www/html
```

Confirmed in deployment model

How It Works

1. EC2 instance launches via Auto Scaling Group
2. User data script executes automatically
3. Instance pulls latest code from S3
4. Application services (Nginx / Apache) start
5. Instance registers with target group and passes health checks

Engineering Advantages

- **Consistency** → Every instance runs identical code
- **Speed** → No manual deployment required
- **Scalability** → New instances are instantly production-ready
- **Resilience** → Failed instances are replaced with correct configuration

This approach directly supports **immutable infrastructure**, where instances are treated as disposable and reproducible.

10.3 User Data Scripts (Frontend & Backend)

Two distinct user data scripts are implemented - one per tier - and embedded within the **Launch Templates**.

As explicitly defined in our architecture

Frontend User Data Script Responsibilities

The frontend script configures the web tier:

- Installs and starts **Nginx**
- Cleans existing web directory
- Pulls frontend code from S3
- Applies Nginx configuration
- Starts web service

Functional Role

- Serves static content (HTML, CSS, JS, images)
- Routes `/api/*` requests to backend via internal ALB

Confirmed in Nginx config behaviour

Backend User Data Script Responsibilities

The backend script configures the application tier:

- Installs **Apache + PHP runtime**
- Pulls backend code from S3
- Applies application and database configuration
- Starts API services

Functional Role

- Processes API requests (`/products`, `/order`)
- Executes business logic
- Interacts with RDS database

Key Design Principle

User data scripts ensure that instances are **self-configuring at launch**, removing any need for manual intervention.

This enables:

- Fully automated scaling
- Rapid recovery from failures
- Environment consistency across AZs

10.4 SSM-Based Deployment Execution

AWS Systems Manager (SSM) is used as the **primary operational control plane** for managing deployments and updates.

Confirmed in SSM architecture design

Why SSM Was Used

- No SSH access required
- Instances are in **private subnets**
- Secure, auditable command execution
- Works via **NAT Gateway outbound access**

Execution Model

SSM Run Command is used to:

- Trigger deployments
- Execute S3 sync commands
- Restart application services

Example operational flow:

SSM Command → EC2 instances
→ Pull latest code from S3
→ Restart services

Connectivity Path

EC2 (Private Subnet)
→ NAT Gateway
→ Internet Gateway
→ AWS Systems Manager

As described in our architecture

Engineering Benefits

- **No bastion host required**
- **No inbound ports exposed**
- Fully aligned with **zero-trust security model**
- Centralised control across all instances

10.5 Update and Release Strategy

The CityMart architecture implements a **safe, controlled, and repeatable release model** based on S3 updates and automated instance refresh mechanisms.

Defined in deployment workflow

Release Workflow

1. Developer updates application code
2. Code uploaded to S3:

citymart-frontend/
citymart-backend/
3. Deployment triggered via:
 - SSM Run Command **OR**
 - Auto Scaling Instance Refresh

Deployment Options

Option 1: SSM-Based Rolling Update

- Push updates to running instances
- Restart services
- Minimal disruption

Option 2: Instance Refresh (Preferred)

- Gradual replacement of instances in ASG
- New instances pull updated code from S3
- Old instances terminated after replacement

Why This Strategy Works

- Ensures **zero downtime deployments**
- Maintains **application consistency**
- Supports **rollback capability** (by reverting S3 version)
- Aligns with **immutable infrastructure principles**

Operational Characteristics

Feature	Implementation
Zero SSH access	SSM Session Manager
Code version control	S3
Deployment trigger	SSM / ASG refresh
Scaling integration	Auto Scaling Groups
Fault recovery	Automatic instance replacement

The CityMart deployment model is a **production-grade, cloud-native deployment pipeline** built on:

- **S3 for centralised artifact storage**
- **Auto Scaling Groups for elastic compute**
- **User data for automated bootstrapping**
- **SSM for secure operational control**

This design achieves:

- Fully automated deployments
- Stateless and reproducible infrastructure
- Secure, no-SSH operations
- Seamless scaling and recovery
- Clean separation between frontend and backend tiers

The CityMart deployment model leverages S3-based artifact storage, Auto Scaling-driven provisioning, user data bootstrapping, and SSM orchestration to deliver a fully automated, secure, and horizontally scalable application deployment pipeline.

11. Database Layer (Amazon RDS)

11.1 Database Architecture Overview

The CityMart platform implements a **managed relational database layer using Amazon RDS (MySQL)** as the persistent data store within the three-tier architecture. This database tier is responsible for storing and retrieving all application-critical data, including:

- Product catalog data
- Customer information
- Order records
- Order item details

The database is logically positioned at the **data layer (Tier 3)** and is accessed exclusively by the backend application tier.

Architectural Placement

The end-to-end data flow is structured as follows:

Frontend EC2 (Nginx)
↓
Backend EC2 (PHP API)
↓
Amazon RDS MySQL (citymart database)

This ensures that:

- The frontend layer **never directly interacts with the database**
- All database operations are mediated through the backend API layer

This design enforces **separation of concerns**, improves **security**, and aligns with best practices for layered architectures

11.2 Subnet Group and Private Deployment

To ensure strong network isolation and controlled access, the database is deployed using a **dedicated DB subnet group**:

- **DB Subnet Group Name:** citymart-database-subnet-group
- **Subnets Used:**
 - db-private-subnet-4a (eu-west-2a)
 - db-private-subnet-4b (eu-west-2b)

aws [Search] [Alt+S] Europe (London) Samuel A. O. (0098-5021-0027) cba_admin

Aurora and RDS > Subnet groups > Create DB subnet group

Create DB subnet group

To create a new subnet group, give it a name and a description, and choose an existing VPC. You will then be able to add subnets related to that VPC.

Subnet group details

Name
You won't be able to modify the name after your subnet group has been created.
citymart-database-subnet-group
Must contain from 1 to 255 characters. Alphanumeric characters, spaces, hyphens, underscores, and periods are allowed.

Description
Citymart database subnet group

VPC
Choose a VPC identifier that corresponds to the subnets you want to use for your DB subnet group. You won't be able to choose a different VPC identifier after your subnet group has been created.
citymart-3-tier-architecture-vpc (vpc-0206bba22a724b50b)
8 Subnets, 2 Availability Zones

Add subnets

Availability Zones
Choose the Availability Zones that include the subnets you want to add.
Choose an availability zone
eu-west-2a X eu-west-2b X

Subnets
Choose the subnets that you want to add. The list includes the subnets in the selected Availability Zones.
Select subnets
db-private-subnet-4a
Subnet ID: subnet-0bcc97ac1cbd99fb6 CIDR: 10.0.144.0/20
db-private-subnet-4b
Subnet ID: subnet-099a048f76a01836f CIDR: 10.0.160.0/20

For Multi-AZ DB clusters, you must select 3 subnets in 3 different Availability Zones.

Subnets selected (2)

Availability zone	Subnet name	Subnet ID	CIDR block
eu-west-2a	db-private-subnet-4a	subnet-0bcc97ac1cbd99fb6	10.0.144.0/20
eu-west-2b	db-private-subnet-4b	subnet-099a048f76a01836f	10.0.160.0/20

Key Design Decisions

1. Private Subnet Deployment

- The database resides **exclusively in private subnets**
- No public IP is assigned
- Public accessibility is explicitly disabled

Public access: NO

2. Network Isolation

- Database is isolated from:
 - Internet traffic
 - Public-facing components
- Only accessible from within the VPC

3. Controlled Routing

- Traffic must traverse:
Backend EC2 → DB Subnet → RDS
- No inbound route from public subnets

Engineering Rationale

This design:

- Minimises attack surface
- Prevents direct exposure of sensitive data
- Enforces **strict internal-only communication**

It aligns with:

- **AWS Security Pillar** → no public exposure
- **Network segmentation best practices**

Confirmed from subnet group + private deployment configuration

11.3 Multi-AZ Configuration

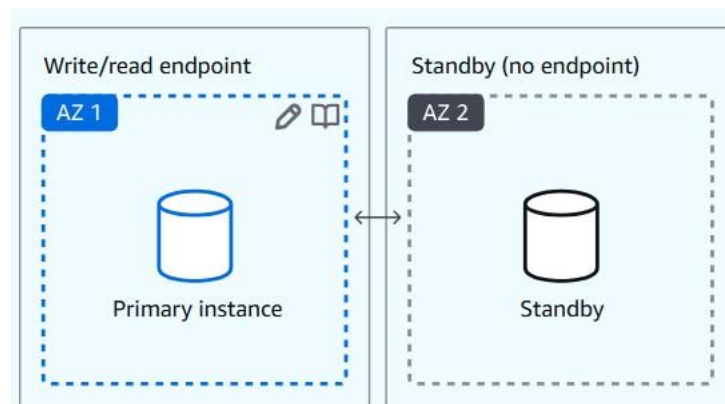
To achieve high availability and fault tolerance, the database is configured using a **Multi-AZ deployment model**.

Implementation Details

- **Primary Instance (AZ-1):** Handles all read/write operations
- **Standby Instance (AZ-2):** Maintains synchronous replica

AZ-1 → Primary (Active)

AZ-2 → Standby (Passive)



How It Works

- Data is **synchronously replicated** from primary to standby

- In the event of failure:
 - Automatic failover occurs
 - Standby instance becomes primary
- The application continues using the **same database endpoint**

Key Benefits

1. High Availability

- Eliminates single point of failure

2. Automatic Failover

- No manual intervention required

3. Data Durability

- Real-time replication ensures minimal or zero data loss

4. Transparent to Application

- Backend continues using same connection endpoint

Engineering Insight

This configuration ensures:

- Continuous service availability
- Resilience against AZ-level outages

Explicit Multi-AZ design with failover capability

11.4 Database Security Model

The CityMart database layer implements a **defense-in-depth security model** combining network-level and application-level controls.

1. Network-Level Security (Primary Control)

Security Groups enforce strict access rules:

- **Database Security Group (citymart-database-sg):**
 - **Inbound:**
 - MySQL (3306) → Allowed **only from Backend EC2 SG**
 - **No public access allowed**

Backend EC2 SG → DB SG (Port 3306)

Traffic Enforcement Chain

Internet
↓
Frontend ALB
↓
Frontend EC2
↓
Internal ALB
↓
Backend EC2
↓
Database (RDS)

Only the backend tier is permitted to communicate with the database.

2. Layer Isolation

Layer	Access Allowed From
Database	Backend EC2 only
Backend EC2	Internal ALB only
Frontend EC2	Public ALB only

This enforces **strict tier isolation**.

3. Security Principles Applied

- Least privilege access
- Zero trust architecture
- No lateral movement between tiers

Fully enforced via security group chaining

11.5 Connectivity and Authentication

Database access is governed by a combination of **network-level connectivity** and **application-level authentication**.

1. Connectivity Model

Backend EC2 instances connect to RDS via:

- Private VPC networking
- Security group-controlled access
- Internal DNS endpoint

Backend EC2 → RDS endpoint (private)

No internet-based access is involved.

2. Authentication Model

Authentication is handled through database credentials defined in application configuration:

- **DB_HOST** → RDS endpoint
- **DB_NAME** → citymart
- **DB_USER** → admin
- **DB_PASS** → password

These credentials are referenced in backend configuration files:

`/var/www/html/config/database.php`

3. Access Control Layers

Layer	Control Mechanism
Network Access	Security Groups
Authentication	DB credentials
Application	Backend API logic

4. Design Implications

- Prevents unauthorised access
- Ensures only authenticated backend services interact with DB
- Maintains separation between infrastructure and application logic

Combined SG + credential-based access control

11.6 Monitoring and Observability

To ensure operational visibility and performance tracking, monitoring capabilities were enabled on the RDS instance.

Monitoring Features Implemented

1. Enhanced Monitoring

- Provides real-time OS-level metrics
- Tracks CPU, memory, and I/O usage

2. CloudWatch Integration

- Enables:
 - Performance metrics

- Resource utilisation tracking
- Alerting capabilities

Observability Benefits

- Detect performance bottlenecks
- Identify abnormal behaviour
- Support troubleshooting and debugging

Alignment with AWS Well-Architected Framework

Pillar	Implementation
Operational Excellence	Monitoring enabled
Performance Efficiency	Metrics tracking
Reliability	Failover visibility

Engineering Insight

Monitoring ensures:

- Proactive issue detection
- Continuous system health validation
- Data-driven performance tuning

Monitoring explicitly enabled for observability

The CityMart database layer is a **secure, highly available, and production-grade RDS deployment** characterised by:

- **Private subnet isolation (no public exposure)**
- **Multi-AZ architecture for fault tolerance**
- **Strict security group-based access control**
- **Backend-only connectivity model**
- **Credential-based authentication**
- **Comprehensive monitoring and observability**

This design fully aligns with AWS best practices across:

- Security
- Reliability
- Performance
- Operational excellence

12. Systems Management and Operations

The CityMart architecture implements a secure, scalable, and operationally efficient systems management model using **AWS Systems Manager (SSM)** as the primary administrative access mechanism. This approach eliminates the need for traditional SSH-based access and aligns with modern cloud security best practices by leveraging **private subnet deployment, IAM-based authentication, and controlled outbound connectivity via NAT Gateways**.

12.1 AWS Systems Manager (SSM)

AWS Systems Manager serves as the **centralised management and control plane** for all EC2 instances deployed within the CityMart architecture.

In this implementation:

- All EC2 instances (frontend and backend) are deployed in **private subnets**
- `ssNo` instance is directly accessible from the public internet
- Administrative access is achieved exclusively through **SSM Session Manager**

From the architecture:

- EC2 instances communicate with AWS Systems Manager via **HTTPS (port 443)**
- Connectivity is enabled through the existing **NAT Gateway infrastructure**

▪ Functional Role of SSM in CityMart

SSM is used to:

- Execute remote commands (e.g., deployments via `aws s3 sync`)
- Access instances securely without SSH
- Manage instances across both frontend and backend Auto Scaling Groups
- Perform operational tasks such as updates, troubleshooting, and configuration validation

▪ Architectural Placement

EC2 (Private Subnet)

↓

NAT Gateway

↓

Internet Gateway

↓

AWS Systems Manager

This design ensures:

- No inbound ports are opened on EC2 instances
- All communication is **outbound-initiated and controlled**

- Management traffic follows the same secure routing path as application updates and S3 access

12.2 Session Manager Access Model

Session Manager provides **interactive shell access** to EC2 instances without requiring:

- SSH keys
- Bastion hosts
- Public IP addresses

This aligns directly with the CityMart architecture design goals of **security and simplicity**.

■ How Session Manager Works

From the implementation:

1. The EC2 instance runs the **SSM Agent**
2. The agent establishes an outbound connection to AWS Systems Manager over HTTPS
3. The administrator initiates a session via AWS Console or CLI
4. The session is securely tunneled through the SSM service

Administrator

↓

AWS Systems Manager

↓

SSM Agent (on EC2)

↓

Instance Shell Access

Confirmed flow from our setup

■ Key Characteristics of the Access Model

1. No SSH Exposure

- Port 22 is not opened on any instance
- No key pair management required

2. No Bastion Host

- Eliminates need for jump servers
- Reduces infrastructure complexity and cost

3. Secure Communication

- All communication occurs over HTTPS (port 443)
- Encrypted and managed by AWS

4. Centralised Access Control

- Access is controlled via IAM policies
- Fully auditable and logged

12.3 IAM Roles and Permissions

IAM plays a critical role in enabling Systems Manager functionality within the CityMart architecture.

Each EC2 instance is assigned an **IAM role (instance profile)** that grants it permission to interact with AWS Systems Manager.

■ **Required IAM Role**

From our implementation:

- Instances are attached to a role containing:
`AmazonSSMManagedInstanceCore`

Explicitly stated in our setup

■ **What This Role Enables**

This managed policy allows EC2 instances to:

- Register with Systems Manager
- Communicate with SSM endpoints
- Receive commands and session requests
- Send logs and execution output

■ **Security Benefits**

1. Least Privilege Access

- Only required permissions are granted
- No excessive IAM privileges

2. No Credential Storage

- No SSH keys or passwords stored on instances
- Authentication handled via IAM

3. Centralised Control

- Access can be granted/revoked at IAM level
- Fine-grained permission management possible

12.4 Private Instance Management Strategy

The CityMart architecture adopts a **private-first infrastructure model**, where all compute resources are isolated from direct internet exposure.

■ Core Design Principles

From our architecture:

- EC2 instances are deployed in **private subnets only**
- No public IP addresses assigned
- All inbound access is blocked except through controlled components (ALBs)

Confirmed in network + security design

■ How Management is Achieved

Instead of inbound access:

- Instances initiate outbound communication via NAT Gateway
- Systems Manager uses this channel to establish control sessions

EC2 (Private)

↓

Private Route Table

↓

NAT Gateway

↓

Internet Gateway

↓

SSM Service

■ Why NAT Gateway is Critical

From our implementation:

- Two NAT Gateways (one per AZ) are deployed for high availability

They enable:

- SSM communication
- S3 access (application code retrieval)
- OS updates and patching

Without NAT Gateway:

- Private instances would have **no external connectivity**
- SSM would not function

▪ Optional Enhancement (Not Implemented)

Our design correctly notes:

- VPC endpoints for SSM are **not required** because NAT is already present
- However, they could be added to:
 - Keep traffic fully within AWS network
 - Improve security posture

Engineering Summary

The CityMart systems management model demonstrates a **modern, secure, and production-grade operational strategy**:

1. Security

- No SSH access
- No public exposure of instances
- IAM-controlled access model

2. Reliability

- Multi-AZ NAT Gateways ensure continuous SSM connectivity
- Instances remain manageable even during scaling events

3. Scalability

- Works seamlessly with Auto Scaling Groups
- Newly launched instances automatically inherit management capabilities

4. Operational Excellence

- Centralised management via SSM
- No manual access setup required
- Supports automation (deployments, updates, monitoring)

The CityMart architecture implements a secure, SSH-less operational model using AWS Systems Manager, where private EC2 instances are managed through IAM-controlled Session Manager access over NAT-enabled outbound connectivity, ensuring centralised control, reduced attack surface, and production-grade operational efficiency.

13. End-to-End System Flow

13.1 Full Request Lifecycle

The CityMart platform implements a structured, multi-layered request lifecycle that ensures secure, scalable, and highly available processing of user interactions across all architectural tiers.

At runtime, every client request follows a deterministic and controlled path through the system, beginning at the global edge layer and terminating at the data layer before returning a response to the user.

Step-by-Step Lifecycle

1. Client Request Initiation

A user accesses the application via:

`https://citymart.samuelonyejekwe.com`

The request is first resolved through the DNS layer and routed to the edge network.

- Route 53 resolves the domain name
- Traffic is directed to CloudFront via an alias record

2. Edge Layer - CloudFront CDN

CloudFront acts as the **global entry point** into the system.

Responsibilities:

- Terminates HTTPS connections
- Serves cached static content where applicable
- Forwards uncached requests to the origin (Frontend ALB)

Connection flow:

User → HTTPS → CloudFront

Key behaviour:

- Static assets (JS, CSS, images) may be served directly from edge locations
- Dynamic requests are forwarded to origin

This reduces latency and offloads backend traffic

3. Origin Layer - Public Application Load Balancer

CloudFront forwards requests to the **frontend ALB**, which serves as the origin.

Connection:

CloudFront → HTTP → Frontend ALB

Responsibilities:

- Distributes incoming traffic across frontend EC2 instances
- Performs health checks
- Ensures high availability across multiple AZs

ALB operates across eu-west-2a and eu-west-2b for fault tolerance

4. Presentation Layer - Frontend EC2 (Nginx)

The request is routed to one of the frontend EC2 instances.

Responsibilities:

- Serve static web content (HTML, CSS, JS)
- Handle user navigation (e.g., /products, /checkout)
- Proxy API calls to backend services

Example:

GET /products.html → served locally

GET /api/products → forwarded to backend

Nginx acts as both web server and reverse proxy

5. Application Routing - Internal ALB

For API requests, Nginx forwards traffic to the internal load balancer:

Frontend EC2 → Internal ALB

Responsibilities:

- Routes API traffic to backend instances
- Ensures backend isolation (not internet accessible)

Backend is fully private and protected from direct access

6. Application Layer - Backend EC2 (PHP API)

Backend servers process business logic.

Responsibilities:

- Handle API endpoints:
 - /api/products
 - /api/product?id=1
 - /api/order
- Perform database queries
- Return structured JSON responses

Example response:

```
{
  "ok": true,
  "data": {
    "products": [...]
  }
}
```

Backend executes core application logic

7. Data Layer - Amazon RDS MySQL

Backend communicates with the database:

Backend EC2 → RDS MySQL

Responsibilities:

- Store persistent data:
 - products
 - customers
 - orders
- Handle read/write operations

Key design:

- Private subnets only
- Multi-AZ deployment for failover

Database is isolated and only accessible from backend tier

8. Response Flow Back to User

The response travels back through the same path:

RDS → Backend → Internal ALB → Frontend → ALB → CloudFront → User

This ensures:

- Consistent routing
- Secure return path
- Optimised delivery via CloudFront

13.2 Component Interaction Sequence

The interaction between components follows a strictly layered and controlled sequence aligned with the three-tier architecture.

Logical Interaction Flow

```
User
↓
Route 53 (DNS)
↓
CloudFront (Edge Layer)
↓
Frontend ALB (Public)
↓
Frontend EC2 (Nginx)
↓
Internal ALB (Private)
↓
Backend EC2 (PHP API)
↓
RDS MySQL
```

This exact sequence is confirmed across multiple architecture diagrams

Interaction Characteristics

1. Layered Communication

Each component communicates **only with the next layer**:

- No direct frontend → database access
- No internet → backend access

Enforced via security groups and subnet isolation

2. Load Balancer Mediation

All traffic is mediated through load balancers:

- Public ALB → frontend distribution

- Internal ALB → backend distribution

Ensures scalability and fault tolerance

3. Stateless Processing

Frontend and backend instances:

- Do not store persistent state
- Can be replaced dynamically by ASG

Enables horizontal scaling and self-healing

4. Controlled API Communication

Frontend communicates with backend via:

/api/*

Handled through:

- Reverse proxy (Nginx)
- Internal ALB routing

Prevents direct exposure of backend services

13.3 Data Flow Across Tiers

The CityMart architecture enforces a **strict and secure data flow model** aligned with the three-tier separation principle.

1. Data Flow Direction

Data flows in a **unidirectional request-response pattern**:

Frontend → Backend → Database → Backend → Frontend

2. Presentation to Application Layer

Frontend sends API requests:

Example:

GET /api/products
POST /api/order

Handled by:

- Nginx proxy
- Internal ALB routing

Frontend never directly accesses database

3. Application to Data Layer

Backend processes and queries database:

SELECT * FROM products
INSERT INTO orders

Security enforcement:

- DB only accessible via backend SG
- Port 3306 restricted

Controlled via security group chaining

4. Data Persistence and Retrieval

Database operations:

- Read product catalog
- Store customer orders
- Maintain transactional data

Multi-AZ ensures data durability and failover

5. Response Data Flow

Processed data is returned as JSON:

Database → Backend → Frontend → User

CloudFront may cache static responses but:

- API responses are NOT cached

Ensures real-time data consistency

6. Security Enforcement in Data Flow

Data movement is controlled at every layer:

Layer	Access Control
Internet → ALB	Public HTTPS
ALB → Frontend	SG restricted
Frontend → Backend	Internal ALB only
Backend → DB	SG + credentials

This creates a **zero-trust, least-privilege architecture**

The CityMart system flow demonstrates a **production-grade, fully layered architecture** where:

- Every request is **routed, validated, and processed through controlled components**
- Each tier is **isolated, scalable, and independently managed**
- Data flows are **secure, structured, and non-bypassable**
- The system achieves:
 - High availability
 - Security
 - Performance efficiency
 - Operational consistency

14. Alignment with AWS Well-Architected Framework

The CityMart three-tier architecture was designed and implemented in alignment with the **AWS Well-Architected Framework**, ensuring that the system meets industry best practices across **operational excellence, security, reliability, performance efficiency, and cost optimisation**.

The architecture demonstrates a production-grade implementation where **CloudFront, Application Load Balancers, Auto Scaling Groups, S3-based deployments, and private networking** are integrated to deliver a resilient and scalable solution .

14.1 Operational Excellence

Operational excellence in the CityMart architecture is achieved through **automation, repeatability, and minimal manual intervention.**

Automated Infrastructure Provisioning

The system leverages:

- **Launch Templates** to define consistent EC2 configurations
- **Auto Scaling Groups (ASGs)** to automatically launch and replace instances

This ensures:

- No configuration drift
- Standardised deployments across all instances

From the implementation:

Instances are launched using launch templates with predefined configurations, ensuring consistency across deployments

Automated Deployment Model (S3 + SSM)

Application deployment is fully automated using:

- **Amazon S3** for centralised code storage
- **SSM Run Command** for remote execution

Deployment workflow:

Upload code → S3 → EC2 pulls via user data or SSM → Service restart

This eliminates:

- Manual SSH access
- Risk of human error

EC2 instances pull application code directly from S3 and updates are triggered via SSM, enabling safe and repeatable deployments

Operational Monitoring & Observability

The architecture integrates:

- **Health checks at ALB level**
- **RDS monitoring via CloudWatch**

Examples:

- `/index.html` → frontend health
- `/api/products` → backend health

This ensures:

- Early detection of failures
- Automatic removal of unhealthy instances

Infrastructure Management Without SSH

The use of:

- **AWS Systems Manager Session Manager**

Provides:

- Secure, auditable access
- No open SSH ports

Instances in private subnets are managed via SSM using IAM roles and outbound connectivity through NAT gateways

Operational Excellence Outcome

- Fully automated deployments
- Self-healing infrastructure
- Minimal operational overhead
- Production-grade maintainability

14.2 Security

Security in the CityMart architecture follows a **defense-in-depth and zero-trust model**, ensuring strict control of access across all layers.

Network Isolation (Core Security Principle)

The architecture enforces strict tier isolation:

Internet → ALB → Frontend → Internal ALB → Backend → Database

- All EC2 instances are in **private subnets**
- Only ALB is publicly accessible

Backend and database layers are not exposed to the internet and are accessible only through controlled internal routing

Security Group Layering (Zero Trust Architecture)

Five security groups enforce strict communication rules:

Layer	Access Allowed
Frontend ALB	Internet
Frontend EC2	Only ALB
Backend ALB	Only frontend
Backend EC2	Only backend ALB
Database	Only backend

Security groups enforce strict tier-to-tier communication and prevent unauthorised access

Encryption in Transit (TLS/HTTPS)

- **ACM certificates** enable HTTPS
- **HTTP → HTTPS redirect enforced at ALB**

Result:

- No plaintext communication from users

HTTPS is enforced using ACM with ALB listeners configured for secure traffic handling

Edge Security with CloudFront

CloudFront provides:

- HTTPS termination
- DDoS protection (AWS Shield)
- Origin protection

CloudFront acts as a secure entry point, preventing direct exposure of backend infrastructure

Database Security

- RDS deployed in **private subnets only**

- Access restricted via:
 - Security groups
 - Credentials

Only backend EC2 instances can access the database on port 3306

Security Outcome

- Zero direct exposure of critical resources
- Enforced least privilege
- End-to-end encrypted communication
- Strong isolation between tiers

14.3 Reliability

Reliability is achieved through **redundancy, fault tolerance, and automated recovery mechanisms**.

Multi-AZ Deployment

All tiers are deployed across:

- **eu-west-2a**
- **eu-west-2b**

This ensures:

- High availability
- Fault tolerance

Infrastructure is distributed across multiple availability zones to prevent single points of failure

Load Balancing Across AZs

- ALBs distribute traffic evenly
- Automatically reroute traffic from failed instances

Auto Scaling Self-Healing

ASGs continuously monitor:

- Desired vs actual capacity

If failure occurs:

Instance fails → terminated → replaced automatically

Auto Scaling ensures automatic replacement of failed instances, maintaining system availability

Database High Availability

- RDS configured in **Multi-AZ mode**
- Automatic failover to standby

If the primary DB fails, traffic is automatically redirected to the standby instance

Health Checks

- Frontend: /index.html
- Backend: /api/products

Ensures:

- Only healthy instances serve traffic

Reliability Outcome

- No single point of failure
- Automatic recovery
- Continuous availability
- Resilient database layer

14.4 Performance Efficiency

Performance efficiency is achieved through **optimised resource usage, intelligent traffic routing, and global content delivery**.

CloudFront Edge Optimisation

CloudFront improves performance by:

- Caching static assets globally
- Serving content from nearest edge location

Static assets such as images and JavaScript files are cached at edge locations to reduce latency

Load Balancing Efficiency

ALBs:

- Distribute traffic evenly
- Prevent overload on individual instances

Stateless Application Design

Frontend and backend tiers are:

- Stateless
- Replaceable

This enables:

- Efficient horizontal scaling
- Faster recovery

Independent Tier Scaling

Frontend and backend scale separately:

- Web traffic spikes handled independently
- Backend processing scaled based on API demand

Independent scaling improves system responsiveness under varying workloads

Optimised Data Access

- Backend interacts directly with RDS
- Private networking reduces latency

Performance Outcome

- Low latency (global + local optimisation)
- Efficient traffic distribution
- Scalable workload handling

14.5 Cost Optimisation

Cost optimisation is achieved through **efficient resource utilisation and on-demand scaling**.

Auto Scaling (Pay-as-you-use)

- Instances scale up only when needed
- Scale down during low traffic

Auto Scaling Groups ensure cost efficiency by adjusting capacity based on demand

Use of Lightweight Instance Types

- `t3.micro` used for both tiers

Benefits:

- Low baseline cost
- Burst performance when needed

CloudFront Cost Efficiency

- Cached content reduces origin requests
- Lower compute usage

S3-Based Deployment

- Cheap, durable storage for application code
- Eliminates need for deployment servers

Private Subnet Design

- Reduces exposure → lowers security-related costs
- Efficient NAT usage

Right-Sized Database

- RDS `db.t3.micro` with `gp3` storage
- Balanced performance and cost

Cost Optimisation Outcome

- Minimal idle resource cost
- Efficient scaling model
- Optimised storage and compute usage

The CityMart architecture demonstrates a **fully aligned AWS Well-Architected implementation**, where:

- **Operational Excellence** → Automation + SSM + S3 deployments
- **Security** → Zero-trust, private tiers, TLS, CloudFront protection
- **Reliability** → Multi-AZ + ASG + RDS failover
- **Performance Efficiency** → CloudFront + ALB + stateless scaling
- **Cost Optimisation** → Auto Scaling + lightweight resources

15. Benefits Achieved

The CityMart three-tier architecture delivers a set of measurable operational, security, and performance benefits aligned with production-grade cloud engineering practices. These benefits are not theoretical; they are directly derived from the implemented AWS services, configurations, and architectural decisions across networking, compute, storage, and delivery layers.

15.1 Improved Customer Experience

The architecture significantly enhances the end-user experience by transforming a manual, phone-based ordering system into a responsive, self-service digital platform.

From the frontend application design, users can:

- Browse product catalogs
- View product details
- Submit order requests via web interface

These interactions are enabled through static frontend content (HTML, CSS, JavaScript) served by Nginx and dynamically enriched via backend API calls such as `/products` and `/order`.

Performance Enhancement via CloudFront

CloudFront plays a critical role in improving user experience by:

- Serving static assets (JS, CSS, images) from edge locations

- Reducing latency by delivering content from the nearest geographic point
- Offloading repeated requests from the origin infrastructure

This ensures faster page load times and smoother interaction, particularly for global users .

Consistent Access via DNS and HTTPS

The integration of Route 53 and ACM ensures:

- Reliable domain resolution (`citymart.samuelonyejekwe.com`)
- Secure HTTPS access for all users
- Elimination of insecure HTTP traffic

This provides a professional, production-ready interface that builds user trust and reliability .

Outcome

The result is a seamless digital shopping experience where users can independently browse and place orders without delays or manual intervention, directly fulfilling the business goal of reducing phone-based operations .

15.2 High Availability and Fault Tolerance

High availability is a core design principle implemented across all layers of the architecture.

Multi-AZ Deployment

All tiers (web, application, and database) are distributed across two Availability Zones:

- eu-west-2a
- eu-west-2b

This ensures that failure in one AZ does not impact overall system availability .

Load Balancing Across Instances

Both frontend and backend tiers are protected by Application Load Balancers:

- Frontend ALB distributes traffic across web servers
- Internal ALB distributes traffic across backend servers

These ALBs continuously monitor instance health and route traffic only to healthy targets .

Auto Scaling Self-Healing Mechanism

Auto Scaling Groups ensure:

- Automatic replacement of failed EC2 instances

- Maintenance of desired capacity
- Continuous service availability without manual intervention

If an instance becomes unhealthy or terminates, it is automatically replaced and re-registered into the target group .

Database Redundancy

The RDS database is configured with Multi-AZ deployment:

- Primary instance in one AZ
- Standby instance in another AZ

This enables automatic failover without changing the database endpoint, ensuring uninterrupted data access.

Outcome

The system achieves strong fault tolerance, eliminating single points of failure and ensuring continuous availability even during infrastructure-level disruptions.

15.3 Scalability and Elasticity

The architecture is designed to scale horizontally and dynamically adapt to workload changes.

Independent Tier Scaling

Frontend and backend layers are managed by separate Auto Scaling Groups:

- Each tier scales independently based on demand
- Resources are provisioned or terminated automatically

This ensures efficient handling of varying workloads without over-provisioning .

Elastic Compute Model

The system demonstrates:

- Scale-out capability (adding more EC2 instances)
- Scale-in capability (reducing instances during low demand)

This elasticity ensures optimal resource utilisation and cost efficiency.

Load Distribution

Application Load Balancers distribute incoming requests evenly across instances, preventing bottlenecks and ensuring consistent performance under load .

CDN Offloading

CloudFront further enhances scalability by:

- Serving cached content at the edge
- Reducing the number of requests reaching origin servers

This significantly reduces backend load during peak traffic scenarios .

Outcome

The system can handle increased traffic seamlessly while maintaining performance, making it suitable for real-world production workloads.

15.4 Security Improvements

Security is enforced through a layered, defense-in-depth architecture.

Network Isolation

The architecture enforces strict tier separation:

- Public access limited to ALB and CloudFront
- Web, application, and database tiers deployed in private subnets

This prevents direct internet exposure of critical resources .

Security Group Chaining

Five dedicated security groups enforce controlled communication:

- Internet → Frontend ALB
- Frontend ALB → Frontend EC2
- Frontend EC2 → Backend ALB
- Backend ALB → Backend EC2
- Backend EC2 → Database

Each layer only accepts traffic from the previous layer, implementing a zero-trust model .

Encryption and Secure Access

- HTTPS enforced via ACM certificates
- TLS termination at ALB and CloudFront
- No plaintext traffic exposed externally

This ensures secure data transmission between clients and the application .

Backend and Database Protection

- Backend servers are not publicly accessible
- Database resides in private subnets with no public access
- Access restricted via security groups and credentials

Operational Security

- No SSH access required
- Instances accessed securely via AWS Systems Manager (Session Manager)

This eliminates exposure of SSH ports and reduces attack vectors.

Outcome

The architecture achieves strong security posture through isolation, encryption, controlled access, and minimised exposure.

15.5 Maintainability and Deployment Efficiency

The system is designed for ease of updates, automation, and operational consistency.

S3-Based Deployment Model

Application code is stored centrally in Amazon S3:

- `citymart-frontend/`
- `citymart-backend/`

Each Auto Scaling Group pulls only the relevant code during instance initialisation.

Automated Instance Configuration

Launch templates ensure:

- Consistent environment setup
- Automated installation of required services (Nginx, Apache, PHP)
- Standardised configuration across all instances

SSM-Based Management

AWS Systems Manager enables:

- Remote command execution
- Secure administration without SSH
- Centralised operational control

Rolling Updates and Refresh Strategy

Updates are performed by:

- Uploading new code to S3
- Triggering instance refresh or SSM deployment

This allows:

- Controlled rollout
- Minimal downtime
- Easy rollback capability

Immutable Infrastructure

Instances are treated as replaceable units:

- No manual configuration drift
- New instances always start from a clean, predefined state

Outcome

The architecture supports efficient operations, rapid deployments, and consistent environments, significantly reducing operational overhead.

Engineering Summary

The CityMart architecture successfully delivers:

- **A user-friendly digital experience**
- **Highly available and fault-tolerant infrastructure**
- **Elastic scalability aligned with demand**
- **Strong, layered security controls**
- **Efficient and automated deployment workflows**

Together, these benefits demonstrate a **production-grade cloud architecture aligned with AWS Well-Architected principles**, validating the system as robust, scalable, and ready for real-world deployment.

16. Conclusion

16.1 Summary of Architecture

The CityMart platform was successfully designed and implemented as a **production-grade three-tier architecture on AWS**, incorporating a fully managed, secure, and scalable cloud-native design.

At a high level, the architecture follows a structured request flow:

User → CloudFront → Public ALB → Frontend EC2 (Nginx) → Internal ALB → Backend EC2 (PHP API) → Amazon RDS (MySQL)

This layered approach enforces **strict separation of concerns across three tiers**:

- **Presentation Tier:** Delivered via Nginx-based frontend servers running in private subnets, serving static content and routing API requests.
- **Application Tier:** Backend EC2 instances executing business logic through PHP APIs, isolated behind an internal load balancer.
- **Data Tier:** A Multi-AZ Amazon RDS MySQL database providing persistent, highly available storage.

The architecture is deployed within a **custom VPC (10.0.0.0/16)** and segmented into **8 subnets across two Availability Zones**, ensuring both **network isolation and high availability**. Public subnets host ingress components such as the Application Load Balancer and NAT Gateways, while all compute and database resources are strictly placed in private subnets.

A key enhancement to the architecture is the integration of **Amazon CloudFront as a global edge layer**, which acts as the primary entry point for user traffic. CloudFront provides **edge caching, latency reduction, and security protection**, while forwarding requests to the public ALB origin.

The solution also incorporates a **fully managed DNS and TLS layer** using Route 53 and AWS Certificate Manager (ACM), enabling secure HTTPS communication and seamless domain resolution.

Compute resources are managed through **Auto Scaling Groups and Launch Templates**, ensuring that both frontend and backend tiers are **horizontally scalable, self-healing, and stateless**. Application deployment is handled through an **S3-based model**, where EC2 instances dynamically pull version-controlled application code at launch, enabling consistent and repeatable deployments.

Security is enforced through a **layered security group model**, where each tier only communicates with its immediate upstream or downstream component, implementing a **zero-trust, least-privilege architecture**.

Overall, the CityMart system represents a **clean, modular, and well-architected cloud solution**, aligned with modern AWS design principles.

16.2 Key Engineering Achievements

The implementation of the CityMart architecture demonstrates several critical engineering achievements that elevate the system from a basic deployment to a **production-grade cloud architecture**.

1. End-to-End Layered Architecture Design

A fully segmented **three-tier architecture** was successfully implemented, ensuring:

- Clear separation between presentation, business logic, and data layers
- Controlled and auditable traffic flow across tiers
- Strong alignment with enterprise architectural patterns

This layered design significantly improves **maintainability, scalability, and security**.

2. High Availability Through Multi-AZ Deployment

All components-including ALBs, EC2 instances, and RDS-were distributed across **two Availability Zones (eu-west-2a and eu-west-2b)**, ensuring:

- Fault tolerance in case of AZ failure
- Continuous traffic distribution via load balancers
- Database failover capability via Multi-AZ RDS

This guarantees **service continuity and resilience under failure conditions** .

3. Horizontal Scalability and Self-Healing Infrastructure

The use of **Auto Scaling Groups integrated with target groups and load balancers** enables:

- Automatic scaling based on demand
- Replacement of unhealthy instances without manual intervention
- Stateless architecture where instances are ephemeral

This achieves true **elastic scalability and operational efficiency** .

4. Secure-by-Design Network Architecture

Security was embedded at every layer of the system through:

- Deployment of all compute and database resources in **private subnets**
- Controlled ingress via a **public Application Load Balancer only**

- Strict **security group chaining** enforcing tier-to-tier communication

Traffic flow is explicitly enforced as:

Internet → ALB → Frontend → Internal ALB → Backend → Database

This eliminates direct exposure of internal resources and significantly reduces the attack surface.

5. Global Performance Optimisation via CloudFront

The integration of **CloudFront as a CDN layer** delivers:

- Reduced latency through edge caching
- Offloading of static content from origin servers
- Secure HTTPS entry point with DDoS protection

This transforms the architecture into a **globally optimised system**, rather than a region-bound deployment .

6. Production-Grade DNS and TLS Implementation

The use of **Route 53 and ACM** enables:

- Fully managed DNS resolution
- HTTPS enforcement with automatic certificate renewal
- Native AWS integration via alias records

This ensures **secure, reliable, and scalable user access** to the application .

7. Automated and Consistent Deployment Model

The architecture employs an **S3-based deployment strategy**, where:

- Application code is centrally stored in S3
- EC2 instances pull code dynamically via user data scripts
- Updates are deployed using SSM or rolling refresh

This results in:

- Consistent environments across instances
- Reduced configuration drift
- Simplified release management

8. Operational Excellence with SSM-Based Access

Administrative access is achieved using **AWS Systems Manager Session Manager**, eliminating the need for SSH access. This:

- Enhances security by removing open ports
- Enables controlled and auditable access
- Aligns with modern cloud operational practices

16.3 Production Readiness Assessment

The CityMart architecture demonstrates a **high level of production readiness**, meeting key criteria across performance, security, reliability, and operational efficiency.

1. Security Readiness

The system enforces:

- HTTPS-only communication via ACM
- Private subnet isolation for all sensitive resources
- Strict security group rules aligned with least privilege

Additionally, CloudFront introduces:

- DDoS protection via AWS Shield
- Secure edge termination

This results in a **defense-in-depth security model suitable for real-world deployment**.

2. Reliability and Fault Tolerance

The architecture achieves strong reliability through:

- Multi-AZ deployment of all tiers
- Load balancing across healthy instances
- Automatic instance replacement via ASGs
- Database failover via Multi-AZ RDS

These features ensure **minimal downtime and continuous service availability**.

3. Scalability Readiness

The system is designed to scale horizontally at both frontend and backend tiers:

- Independent scaling of web and application layers
- Load balancer integration with target groups
- Stateless compute design

This enables the system to handle **variable workloads efficiently without manual intervention**.

4. Performance Optimisation

Performance is enhanced through:

- CloudFront edge caching for static assets
- Compression (Gzip/Brotli)
- Efficient request routing via ALBs

This ensures **low latency and fast response times for end users globally**.

5. Operational and Deployment Readiness

The use of:

- Launch templates
- S3-based deployments
- SSM for management

ensures that the system is:

- Easy to maintain
- Easy to update
- Fully automated

This aligns with **DevOps and infrastructure-as-code principles**, even within a manual deployment context.

6. Alignment with AWS Well-Architected Framework

The solution demonstrates alignment with key pillars:

- **Operational Excellence:** Automated deployments and SSM-based management
- **Security:** Private subnets, SG chaining, HTTPS enforcement
- **Reliability:** Multi-AZ, ASG self-healing, health checks
- **Performance Efficiency:** CloudFront + ALB + horizontal scaling

- **Cost Optimisation:** t3.micro instances and dynamic scaling

Final Assessment

The CityMart system can be confidently classified as a:

Production-ready, cloud-native, highly available, and scalable 3-tier architecture

It successfully transforms a simple retail requirement into a **robust, secure, and professionally engineered cloud solution**, demonstrating strong practical application of AWS architectural best practices.

References

Amazon Web Services (AWS) (n.d.) *Amazon CloudFront Developer Guide*. Available at: <https://docs.aws.amazon.com/cloudfront/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *Amazon Route 53 Developer Guide*. Available at: <https://docs.aws.amazon.com/route53/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *Elastic Load Balancing Application Load Balancer Guide*. Available at: <https://docs.aws.amazon.com/elasticloadbalancing/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *Amazon EC2 Auto Scaling User Guide*. Available at: <https://docs.aws.amazon.com/autoscaling/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *Amazon EC2 User Guide for Linux Instances*. Available at: <https://docs.aws.amazon.com/ec2/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *Amazon RDS for MySQL User Guide*. Available at: <https://docs.aws.amazon.com/rds/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *Amazon VPC User Guide*. Available at: <https://docs.aws.amazon.com/vpc/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *AWS Certificate Manager User Guide*. Available at: <https://docs.aws.amazon.com/acm/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *Amazon S3 User Guide*. Available at: <https://docs.aws.amazon.com/s3/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *AWS Systems Manager User Guide*. Available at: <https://docs.aws.amazon.com/systems-manager/> (Accessed: April 2026).

Amazon Web Services (AWS) (n.d.) *AWS Well-Architected Framework*. Available at: [sssssshttps://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html](https://docs.aws.amazon.com/wellarchitected/latest/framework/welcome.html) (Accessed: April 2026).