SOFTWARE DESIGN DOCUMENT

(SDD)

Cloudimart E-Commerce Platform

Technical Architecture and Implementation Design

Version 1.0
February 2026

Prepared by:
Christopher R. Kuchawo

Cloudimart Limited

TABLE OF CONTENTS

1. INTRODUCTION

1.1 Purpose

This Software Design Document (SDD) describes the technical architecture, detailed design, and implementation strategy for the Cloudimart E-Commerce Platform. It serves as the primary technical reference for developers, system architects, and technical stakeholders involved in the construction, deployment, and maintenance of the system.

The document translates the requirements specified in the Software Requirements Specification (SRS) into concrete technical designs, including system architecture, database schema, API specifications, algorithms, and component interactions. It provides sufficient detail to guide implementation while maintaining flexibility for technical optimization during development.

1.2 Scope

This document covers the complete technical design of the Cloudimart platform, including:

- System architecture and component organization
- Database schema with tables, relationships, and constraints
- Backend service layer design using Laravel framework
- Frontend component architecture using Next.js
- RESTful API endpoint specifications
- Core algorithms (geofencing, order ID generation, verification)
- Security mechanisms (authentication, authorization, encryption)
- Integration interfaces with external services (SMS, email, GPS)
- Deployment architecture and infrastructure requirements

This document does NOT cover: detailed user interface mockups (separate UI/UX documentation), quality assurance test plans (separate QA documentation), operational procedures and runbooks (separate DevOps documentation), or business process workflows (covered in SRS).

1.3 Document Overview

The document is organized into seven main sections:

Section 1 (Introduction): Establishes document purpose, scope, and conventions

Section 2 (System Architecture): Describes high-level architecture, layers, and technology choices

Section 3 (Detailed Design): Provides comprehensive database design, class structures, and component specifications

Section 4 (System Algorithms): Documents core algorithms with pseudocode and complexity analysis

Section 5 (Interface Design): Specifies user interfaces, API contracts, and external integrations

Section 6 (Security Design): Details security mechanisms, authentication flows, and data protection

Section 7 (Deployment): Describes deployment architecture, infrastructure, and scaling strategies

1.4 Definitions and Acronyms

Technical terms and acronyms used in this document (additional to those in SRS):

DTO: Data Transfer Object - object carrying data between processes

Repository Pattern: Design pattern mediating between domain and data mapping layers

Migration: Versioned database schema change script

Seeder: Script populating database with initial or test data

Middleware: Software layer processing requests before reaching controllers

Service Layer: Business logic layer between controllers and models

Eloquent: Laravel's ORM implementation for database interactions

Blade: Laravel's templating engine (not used in Next.js frontend)

Server-Side Rendering (SSR): Rendering web pages on server before sending to client

Static Generation: Pre-rendering pages at build time

2. SYSTEM ARCHITECTURE

2.1 Architectural Overview

Cloudimart employs a three-tier client-server architecture with clear separation of concerns between presentation, business logic, and data persistence layers. The architecture follows REST principles for API communication and MVC pattern for backend organization.

Key Architectural Principles:

- Separation of Concerns: Clear boundaries between UI, business logic, and data access
- Modularity: Independent, loosely-coupled components with well-defined interfaces
- Scalability: Stateless API design enabling horizontal scaling
- Security by Design: Authentication, authorization, and encryption at every layer
- Fault Tolerance: Graceful degradation and error handling throughout
- Maintainability: Consistent coding standards, comprehensive documentation, automated testing

2.2 Logical Architecture

The system is organized into four logical layers with specific responsibilities:

2.2.1 Presentation Layer (Frontend)

Technology: Next.js 14 with React 18, TypeScript for type safety

Responsibilities: Render user interfaces, capture user input, handle client-side validation, manage application state, communicate with API via HTTP

Key Components: Page components (Home, Products, Cart, Checkout, Orders), Reusable UI components (ProductCard, CartItem, LocationSelector), Context providers (AuthContext, CartContext), Utility hooks (useGeolocation, useCart, useAuth)

State Management: React Context API for global state (cart, authentication), component state for local UI state

Routing: Next.js file-based routing with dynamic routes for product details

2.2.2 API Layer (Middleware)

Technology: Laravel 10 routing and middleware stack

Responsibilities: Route HTTP requests, authenticate users, authorize actions, validate input, handle CORS, manage sessions, log requests

Key Middleware: Authentication (Laravel Sanctum), CORS handling, Request validation, Rate limiting, Geofence validation

Response Format: JSON responses with consistent structure: {success: boolean, data: object, message: string, errors: array}

2.2.3 Application Layer (Business Logic)

Technology: Laravel 10 MVC components (Controllers, Services, Models)

Responsibilities: Implement business rules, process transactions, coordinate operations, orchestrate service interactions

Controllers: ProductController, OrderController, CartController, DeliveryController, AuthController - handle HTTP request/response cycle

Services: GeofenceService (validation logic), OrderService (order processing), NotificationService (email/SMS), PaymentService (payment simulation)

Models: Eloquent models (User, Product, Category, Order, OrderItem, DeliveryLocation, Delivery) - represent database entities and relationships

2.2.4 Data Persistence Layer

Technology: MySQL 8.0 with InnoDB storage engine

Responsibilities: Store and retrieve data, enforce referential integrity, provide transaction support, optimize query performance

Access Method: Eloquent ORM with query builder for complex queries

Key Tables: users, products, categories, orders, order_items, carts, cart_items, delivery_locations, deliveries, migrations

2.3 Physical Architecture

The physical deployment consists of four primary components deployed on cloud or on-premise infrastructure:

- Web Server:Nginx serving Next.js static files and proxying API requests to application server. Handles SSL/TLS termination, load balancing, static asset serving.
- Application Server:PHP-FPM running Laravel application. Processes business logic, generates dynamic content, coordinates with database and external services.
- Database Server:MySQL server with master configuration (replica for scaling if needed). Stores all persistent application data with automated backups.
- External Services:TNM SMS Gateway (cloud), Email Service/Mailgun (cloud), DNS and CDN (cloud)

2.4 Technology Stack

Comprehensive technology stack with versions and justifications:

2.4.1 Frontend Stack

Next.js 14: React framework providing server-side rendering, static generation, optimized routing, and excellent developer experience

React 18: Component-based UI library for building interactive interfaces

TypeScript 5: Typed superset of JavaScript improving code quality and developer productivity

Tailwind CSS 3: Utility-first CSS framework for rapid UI development

Axios: Promise-based HTTP client for API communication

React Hook Form: Performant, flexible forms with easy validation

2.4.2 Backend Stack

Laravel 10: Full-featured PHP framework with elegant syntax, robust ecosystem, and comprehensive documentation

PHP 8.2: Server-side scripting language with modern features (typed properties, enums, attributes)

Laravel Sanctum: Lightweight authentication for SPAs and mobile apps using token-based auth

Guzzle HTTP: PHP HTTP client for making requests to external APIs

Laravel Queue: Queue system for handling asynchronous tasks (email/SMS sending)

2.4.3 Database and Infrastructure

MySQL 8.0: Reliable, performant relational database with JSON support for flexible data storage

Redis: In-memory data store for caching, sessions, and queue backend

Nginx 1.20+: High-performance web server and reverse proxy

Docker: Containerization platform ensuring consistent environments

Ubuntu 22.04 LTS: Stable, secure Linux distribution for server deployment

3. DETAILED DESIGN

3.1 Database Design

The database schema follows third normal form (3NF) to minimize redundancy while maintaining query performance through strategic denormalization where justified. All tables use InnoDB engine for transaction support and foreign key constraints.

3.1.1 Core Entity Tables

users Table

- id: BIGINT UNSIGNED PRIMARY KEY AUTO_INCREMENT
- name: VARCHAR(255) NOT NULL
- email: VARCHAR(255) UNIQUE NOT NULL
- phone: VARCHAR(20) NOT NULL
- password: VARCHAR(255) NOT NULL (bcrypt hashed)
- role: ENUM("customer", "admin", "delivery_staff") DEFAULT "customer"
- created_at, updated_at: TIMESTAMP
- INDEX: email, phone

products Table

- id: BIGINT UNSIGNED PRIMARY KEY AUTO_INCREMENT
- category_id: BIGINT UNSIGNED FOREIGN KEY REFERENCES categories(id)
- name: VARCHAR(255) NOT NULL
- description: TEXT
- price: DECIMAL(10,2) NOT NULL
- stock_quantity: INT UNSIGNED NOT NULL DEFAULT 0
- image_url: VARCHAR(500)
- is_active: BOOLEAN DEFAULT TRUE
- created_at, updated_at: TIMESTAMP
- INDEX: category_id, is_active, name

orders Table

- id: BIGINT UNSIGNED PRIMARY KEY AUTO_INCREMENT
- order_id: VARCHAR(50) UNIQUE NOT NULL (ORD-YYYYMMDD-XXX format)
- user_id: BIGINT UNSIGNED FOREIGN KEY REFERENCES users(id)
- delivery_location_id: BIGINT UNSIGNED FOREIGN KEY REFERENCES delivery_locations(id)
- total_amount: DECIMAL(10,2) NOT NULL
- status: ENUM("pending", "processing", "out_for_delivery", "delivered", "cancelled") DEFAULT "pending"
- payment_status: ENUM("pending", "completed", "failed") DEFAULT "pending"
- created_at, updated_at: TIMESTAMP

- INDEX: order_id (UNIQUE), user_id, status, created_at

delivery_locations Table

- id: BIGINT UNSIGNED PRIMARY KEY AUTO_INCREMENT
- name: VARCHAR(255) NOT NULL (e.g., "Mzuzu University", "Luwinga")
- polygon_coords: JSON NOT NULL (array of {lat, lng} points defining boundary)
- is_active: BOOLEAN DEFAULT TRUE
- created_at, updated_at: TIMESTAMP
- INDEX: name, is_active

3.2 Class and Service Design

The application layer implements business logic through controllers coordinating with service classes and Eloquent models. This separation enables testability, reusability, and maintainability.

3.2.1 GeofenceService Class

Purpose: Validate user coordinates against delivery zone polygons using point-in-polygon algorithm

Key Methods:

- validateLocation(float $latitude, float $longitude): array - Returns {isValid: bool, zoneName: string|null, nearestZone: string|null}
- isPointInPolygon(array $point, array $polygon): bool - Implements ray-casting algorithm
- getActiveDeliveryZones(): Collection - Retrieves all active delivery zones from database
- findNearestZone(float $latitude, float $longitude): string|null - Finds closest zone to given coordinates

3.2.2 OrderService Class

Purpose: Handle order creation, Order ID generation, and order lifecycle management

Key Methods:

- generateOrderId(): string - Creates unique Order ID in ORD-YYYYMMDD-XXX format
- createOrder(User $user, Cart $cart, DeliveryLocation $location): Order - Creates order with items
- updateOrderStatus(Order $order, string $newStatus): bool - Updates status with validation
- cancelOrder(Order $order): bool - Cancels order within allowed timeframe

4. SYSTEM ALGORITHMS

4.1 Geofence Validation Algorithm

The geofence validation uses the ray-casting algorithm (Jordan curve theorem) to determine whether a point lies inside a polygon. This algorithm is reliable, efficient (O(n) where n = polygon vertices), and handles complex polygons including concave shapes.

Algorithm Pseudocode:

```
function isPointInPolygon(point, polygon):
    lat = point.latitude
    lng = point.longitude
    inside = false

    for each edge in polygon:
        vertex1 = edge.start
        vertex2 = edge.end

        if ((vertex1.lng > lng) != (vertex2.lng > lng)) and
           (lat < (vertex2.lat - vertex1.lat) * (lng - vertex1.lng) /
                  (vertex2.lng - vertex1.lng) + vertex1.lat):
            inside = !inside

    return inside
```

Time Complexity: O(n) where n is the number of polygon vertices

Space Complexity: O(1) - constant space for variables

4.2 Order ID Generation Algorithm

Order ID generation combines date-based prefixing with database-driven sequential numbering to ensure uniqueness while maintaining human-readable format. The algorithm uses database transactions to prevent race conditions in concurrent order creation.

Algorithm Pseudocode:

```
function generateOrderId():
    BEGIN TRANSACTION

    currentDate = getCurrentDate() // YYYY-MM-DD format
    datePrefix = "ORD-" + format(currentDate, "YYYYMMDD")

    // Get today's order count
    todayStart = currentDate + " 00:00:00"
    todayEnd = currentDate + " 23:59:59"

    orderCount = SELECT COUNT(*) FROM orders
                 WHERE created_at BETWEEN todayStart AND todayEnd

    sequenceNumber = orderCount + 1
    paddedSequence = padLeft(sequenceNumber, 3, '0') // e.g., "001", "042", "123"
    orderId = datePrefix + "-" + paddedSequence

    // Verify uniqueness (should always pass due to transaction isolation)
    if EXISTS(SELECT 1 FROM orders WHERE order_id = orderId):
        // Handle edge case - unlikely but possible
        paddedSequence = padLeft(sequenceNumber + 1, 3, '0')
        orderId = datePrefix + "-" + paddedSequence

    COMMIT TRANSACTION
    return orderId
```

4.3 Delivery Handshake Verification Algorithm

Two-factor verification prevents fraudulent delivery confirmations by requiring both Order ID and customer phone number to match database records. The algorithm includes rate limiting to prevent brute-force attacks.

Algorithm Pseudocode:

```
function verifyDeliveryHandshake(orderId, customerPhone, deliveryStaffId):
    // Rate limiting check
    recentAttempts = getRecentVerificationAttempts(deliveryStaffId, last5Minutes)
    if recentAttempts > 10:
        return {success: false, error: "Too many attempts, please wait"}

    // Retrieve order
    order = SELECT * FROM orders WHERE order_id = orderId

    if order is NULL:
        logFailedAttempt(deliveryStaffId, orderId, "Order not found")
        return {success: false, error: "Invalid Order ID"}

    // Get customer phone from user record
    customer = SELECT * FROM users WHERE id = order.user_id

    // Normalize phone numbers (remove spaces, dashes, country codes)
    normalizedInput = normalizePhone(customerPhone)
    normalizedStored = normalizePhone(customer.phone)

    if normalizedInput != normalizedStored:
        logFailedAttempt(deliveryStaffId, orderId, "Phone mismatch")
        return {success: false, error: "Phone number does not match order"}

    // Both factors match - proceed with delivery confirmation
    BEGIN TRANSACTION

    UPDATE orders
    SET status = 'delivered',
        updated_at = NOW()
    WHERE id = order.id

    INSERT INTO deliveries (order_id, delivery_person_id, customer_phone, status,
delivered_at)
    VALUES (order.id, deliveryStaffId, customerPhone, 'completed', NOW())
```

```
// Send confirmation notification
sendNotification(customer.email, customer.phone, "Order delivered successfully")

COMMIT TRANSACTION

logSuccessfulDelivery(deliveryStaffId, orderId)
return {success: true, message: "Delivery confirmed"}
```

5. INTERFACE DESIGN

5.1 User Interface Design Patterns

The user interface follows mobile-first responsive design principles with consistent component patterns across all screens. Key UI patterns include:

Card-based Product Display: Products displayed in grid cards (2 columns mobile, 3-4 desktop) showing image, name, price, stock status, and add-to-cart button

Sticky Shopping Cart Icon: Floating cart icon in header showing item count, persists across pages

Progressive Disclosure: Complex forms (checkout) broken into logical steps with progress indicator

Toast Notifications: Non-intrusive notifications for actions (item added, order placed) appearing bottom-right, auto-dismissing after 3 seconds

Loading Skeletons: Skeleton screens during data loading providing visual feedback and perceived performance improvement

Empty States: Helpful empty state messages when no data (empty cart, no orders) with call-to-action buttons

5.2 RESTful API Endpoints

The API follows REST principles with resource-based URLs, HTTP verbs for actions, and JSON request/response bodies. All endpoints require HTTPS and return consistent response format.

Core API Endpoints:

Product Management:

- GET /api/products - List products with pagination, filtering, sorting
- GET /api/products/{id} - Get single product details
- POST /api/products - Create new product (Admin only)
- PUT /api/products/{id} - Update product (Admin only)
- DELETE /api/products/{id} - Delete product (Admin only)
- GET /api/categories - List all product categories

Shopping Cart:

- GET /api/cart - Get current user's cart
- POST /api/cart/items - Add item to cart
- PUT /api/cart/items/{id} - Update cart item quantity
- DELETE /api/cart/items/{id} - Remove item from cart
- DELETE /api/cart - Clear entire cart

Order Processing:

- POST /api/checkout - Process checkout with geofence validation
- GET /api/orders - List user's orders
- GET /api/orders/{orderId} - Get order details
- PATCH /api/orders/{orderId}/cancel - Cancel order
- GET /api/admin/orders - List all orders (Admin only)

Delivery Verification:

- GET /api/delivery/assigned - Get assigned deliveries for delivery staff
- POST /api/delivery/verify - Verify delivery handshake
- GET /api/delivery/history - View delivery history

Geofence Validation:

- POST /api/geofence/validate - Validate coordinates against delivery zones
- GET /api/delivery-zones - List active delivery zones (Admin)

6. SECURITY DESIGN

6.1 Authentication Mechanism

Authentication uses Laravel Sanctum providing token-based authentication suitable for SPAs and mobile apps. The system implements secure registration, login, and session management.

Authentication Flow:

- 1. User submits credentials (email/phone + password) to /api/login endpoint
- 2. Backend validates credentials against bcrypt-hashed password in database
- 3. If valid, server generates Sanctum access token (random 40-character string)
- 4. Token stored in personal_access_tokens table with user_id, expiration (24 hours)
- 5. Token returned to client, stored in HTTP-only cookie or local storage
- 6. Subsequent requests include token in Authorization: Bearer {token} header
- 7. Middleware validates token on each protected request
- 8. Token can be revoked via logout endpoint or expires after 24 hours inactivity

6.2 Authorization Model (RBAC)

Role-Based Access Control (RBAC) restricts system access based on user roles. Three primary roles with distinct permissions:

Role	Permissions	Restrictions
Customer	Browse products, manage cart, place orders, view own orders, update profile	Cannot access admin features, cannot modify products, cannot view other users' data
Admin	All customer permissions + manage products/categories, configure delivery zones, view all orders, manage users	Cannot impersonate other users, changes logged for audit
Delivery Staff	View assigned deliveries, verify delivery handshake, update delivery status	Cannot place orders, cannot access admin functions, cannot view unassigned deliveries

6.3 Data Protection Measures

Password Hashing: All passwords hashed using Bcrypt (cost factor 10) before storage. Plain-text passwords never stored or logged.

HTTPS/TLS: All communications encrypted using TLS 1.2+ with strong cipher suites. HTTP redirects to HTTPS.

Input Sanitization: All user inputs sanitized using Laravel's built-in validation and Eloquent ORM preventing SQL injection. Output escaped to prevent XSS.

CSRF Protection: All state-changing requests require valid CSRF token verified by middleware.

Rate Limiting: API endpoints rate-limited (60 requests/minute per IP for auth, 120/minute for general APIs).

SQL Injection Prevention: Parameterized queries via Eloquent ORM. No raw SQL with user input.

Sensitive Data Logging: Passwords, tokens, payment info excluded from application logs.

7. DEPLOYMENT ARCHITECTURE

7.1 Deployment Overview

The system deploys in containerized environment using Docker for consistency across development, staging, and production environments. The deployment consists of multiple containers orchestrated via Docker Compose.

Web Container (Nginx): Serves static Next.js files, proxies API requests to app container, handles SSL termination. Image: nginx:1.21-alpine. Port: 80 (HTTP), 443 (HTTPS).

App Container (Laravel): Runs PHP-FPM with Laravel application. Processes business logic, connects to database and cache. Image: php:8.2-fpm-alpine with extensions. Port: 9000 (FastCGI).

Database Container (MySQL): Stores persistent application data. Configured with UTF8mb4 charset, InnoDB engine. Image: mysql:8.0. Port: 3306. Volume: /var/lib/mysql for data persistence.

Cache Container (Redis): Provides caching layer, session storage, queue backend. Image: redis:7-alpine. Port: 6379.

Queue Worker Container: Processes background jobs (email/SMS sending). Same image as app container running queue:work command.

7.2 Infrastructure Requirements

Minimum and Recommended Infrastructure Specifications:

Component	Minimum	Recommended
CPU	2 cores @ 2.0 GHz	4 cores @ 2.5 GHz
RAM	4 GB	8 GB
Storage	50 GB SSD	100 GB SSD
Network	100 Mbps	1 Gbps
Bandwidth	500 GB/month	2 TB/month

7.3 Scalability Considerations

The architecture supports both vertical and horizontal scaling to accommodate growth:

Horizontal Scaling (Web/App Tiers): Deploy multiple app containers behind load balancer (Nginx, HAProxy, or cloud load balancer). Stateless API design enables seamless distribution of requests.

Database Scaling: Vertical scaling (more CPU/RAM) for initial growth. Horizontal read scaling via MySQL replication (master-replica). Connection pooling to optimize database connections.

Caching Strategy: Redis caching for frequently-accessed data (products, categories). Query result caching with TTL. CDN for static assets (images, CSS, JS).

Queue Processing: Multiple queue worker containers for background job processing. Separate queues for different priority levels.

Monitoring and Auto-Scaling: Application performance monitoring (APM) tools. Container orchestration (Kubernetes) for auto-scaling based on CPU/memory metrics.

APPENDIX A: DATABASE SCHEMA REFERENCE

Complete database schema with all tables, columns, data types, constraints, and indexes. Refer to migration files in /database/migrations/ directory for authoritative schema definitions.

APPENDIX B: API SPECIFICATION REFERENCE

Complete API documentation following OpenAPI 3.0 specification available at /api/documentation endpoint when application is running. Interactive API explorer (Swagger UI) available for testing endpoints.

--- END OF DOCUMENT ---