



# EAST WEST UNIVERSITY

## **Multi-Tenant Commerce Marketplace “Bazario” Project Report**

### **Submitted To**

**Yasin Sazid**

Lecturer

Department of Computer Science and Engineering

East West University

**Course Code:** CSE 412

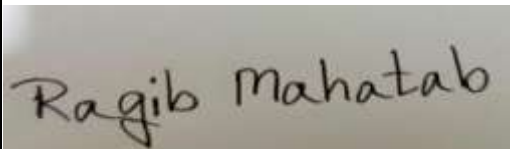
**Section:** 03

**Date:** 03 September 2025

### **Submitted By (Group 07):**

Farhana Ahmed Tasnim	ID: 2022-1-60-156
Md Sifat Ullah Sheikh	ID: 2022-1-60-029
Apurbo Chandra Paul	ID: 2020-3-60-092
Ragib Mahatab	ID: 2021-1-60-050

**Table 1.1: Group Member Information**

<b>Name</b>	<b>ID</b>	<b>Signature</b>
Md Sifat Ullah Sheikh	2022-1-60-029	
Farhana Ahmed Tasnim	2022-1-60-156	
Apurbo Chandra Paul	2020-3-60-092	
Ragib Mahatab	2021-1-60-050	

# Table of Contents

<b>Chapter 1. Introduction .....</b>	<b>6</b>
1.1 Project Overview .....	6
1.2 Objectives and Scope .....	6
1.3 Stakeholders .....	7
1.4 Technology Stack .....	8
<b>Chapter 2. Software Requirements Specification &amp; Analysis ....</b>	<b>9</b>
2.1 Stakeholder Needs & Analysis .....	9
2.1.1 Primary and Secondary Stakeholders .....	9
2.1.2 Methods for Requirement Elicitation .....	10
2.2 List of Requirements .....	11
2.2.1 Functional Requirements (FRs) .....	6
2.2.2 Non-Functional Requirements (NFRs) .....	7
2.2.3 Extra-Ordinary Requirements (Wow Factors) .....	8
2.3 Quality Function Deployment .....	9
2.3.1 Customer Requirements (CRs) .....	9
2.3.2 Engineering Requirements (TRs) .....	10
2.3.3 QFD Matrix (House of Quality) .....	11
2.4 Requirements Modeling .....	12
2.4.1 Use Case Diagrams .....	12
2.4.2 Activity Diagrams .....	13
2.4.3 UI Sketches .....	14
<b>Chapter 3. Software Design .....</b>	<b>15</b>
3.1 Architectural Design .....	19
3.1.1 Architectural Context Diagram .....	20
3.1.2 Top-Level Component Diagram .....	21
3.1.3 Component Elaboration .....	22
3.2 Component-Level Design .....	23
3.2.1 Design Components .....	24
3.2.2 Class Diagram .....	25
3.2.3 Database Design .....	26
3.3 User Interface Design .....	27
3.3.1 UI Wireframes/Mockups .....	28
3.3.2 Navigation Flow .....	29
<b>Chapter 4. Implementation .....</b>	<b>30</b>
4.1 Code Structure Overview .....	31
4.2 GitHub Repository URL .....	32

<b>Chapter 5. Software Testing</b> .....	33
5.1 White Box Testing .....	33
5.1.1 Unit Testing on Two Classes .....	34
5.2 Black Box Testing .....	35
5.2.1 Functional Testing on Two Core Features .....	36
5.3 Bug Detection and Solution .....	37
5.3.1 Identification of Bugs .....	37
5.3.2 Solutions and Retesting .....	38
 <b>Chapter 6. Deployment</b> .....	44
6.1 Deployment Platform .....	44
6.2 Deployment Process .....	44
6.3 Website URL .....	45
 <b>Chapter 7. Conclusion</b> .....	46
7.1 Learnings .....	46
7.2 Limitations .....	46
7.3 Future Plan .....	47

## List of Figures

• <b>Figure 1.1:</b> Use Case Diagram for Bazario System .....	13
• <b>Figure 1.2:</b> Level 0 for Multi-Vendor E-Commerce Marketplace .....	13
• <b>Figure 1.3:</b> Level 1 for Multi-Vendor E-Commerce Marketplace .....	14
• <b>Figure 1.4:</b> Level 2.1 for Multi-Vendor E-Commerce Marketplace .....	15
• <b>Figure 1.5:</b> Level 2.2 for Multi-Vendor E-Commerce Marketplace .....	16
• <b>Figure 1.6:</b> Level 2.3 for Multi-Vendor E-Commerce Marketplace .....	17
• <b>Figure 1.7:</b> Level 2.4 for Multi-Vendor E-Commerce Marketplace .....	18
• <b>Figure 1.8:</b> Level 2.5 for Multi-Vendor E-Commerce Marketplace .....	19
• <b>Figure 1.9:</b> Level 2.6 for Multi-Vendor E-Commerce Marketplace .....	20
• <b>Figure 1.10:</b> Level 2.7 for Multi-Vendor E-Commerce Marketplace .....	21
• <b>Figure 2:</b> Activity Diagram for Multi-Vendor E-Commerce Marketplace .....	22
• <b>Figure 3.1:</b> Products Interface for Users (Homepage View) .....	24
• <b>Figure 3.2:</b> Products Interface for Users (Category Filtering) .....	25
• <b>Figure 3.3:</b> Products Interface for Users (Cart and Checkout) .....	26
• <b>Figure 3.4:</b> Products Interface for Users (Vendor Dashboard) .....	27
• <b>Figure 4:</b> Architectural Context Diagram .....	29
• <b>Figure 5:</b> Top-Level Component Diagram of the System .....	30
• <b>Figure 6:</b> Class Diagram of Core Entities .....	32
• <b>Figure 7:</b> Database ER Diagram of the Marketplace .....	34
• <b>Figure 8.1:</b> Wireframe of the Homepage .....	35
• <b>Figure 8.2:</b> Admin Dashboard .....	35
• <b>Figure 9.1:</b> Navigation Flow of the System .....	36
• <b>Figure 10.1:</b> Paths Covered for Login Module .....	36
• <b>Figure 10.2:</b> Decision-to-Decision Graph for Login Module .....	36
• <b>Figure 10.3:</b> Cyclomatic Complexity for Login Module .....	36
• <b>Figure 11.1:</b> Decision-to-Decision Graph for Register Module .....	36
• <b>Figure 11.2:</b> Cyclomatic Complexity for Register Module .....	37
• <b>Figure 12:</b> Decision-to-Decision Graph for Confirm Order Module .....	39
• <b>Figure 13.1:</b> Customer Registration Module Bug and Fix .....	42
• <b>Figure 13.2:</b> Summary of Payment Bugs, Fixes, and Re-test Results .....	43

## List of Tables

• <b>Table 1.1:</b> Quality Function Deployment (QFD) Matrix for Bazario .....	12
• <b>Table 2.1:</b> Black Box Test Cases – Customer Registration .....	38
• <b>Table 2.2:</b> Black Box Test Cases – Payment Confirmation .....	39

# Chapter 1: Introduction

## 1.1 Project Overview

The *Multi-Vendor E-Commerce Marketplace* is an online platform designed to bring together multiple vendors and buyers within a unified digital environment. In today's fast-paced digital economy, online shopping has emerged as the dominant mode of commerce, enabling convenience and accessibility for customers worldwide. Unlike traditional single-vendor platforms, a multi-vendor marketplace provides a centralized system where different sellers can register, showcase their products, and reach a broader audience simultaneously.

This project seeks to develop a robust, user-friendly, and scalable marketplace system that accommodates the diverse needs of vendors, buyers, and administrators. Vendors will have access to tools for product management and order processing, buyers will benefit from seamless browsing and secure transactions, while administrators will oversee platform governance and ensure compliance.

## 1.2 Objectives and Scope

The primary objective of this project is to design and implement a fully functional e-commerce marketplace that supports multiple stakeholders with tailored features. The specific objectives include:

- To develop a **vendor management system** that facilitates easy onboarding and product listing.
- To provide buyers with a **secure and reliable shopping experience**, including product search, cart management, and checkout.
- To equip administrators with **management and analytics tools** for effective monitoring and decision-making.
- To ensure the system is **scalable, reliable, and user-friendly**, capable of handling growth in both users and transactions.

### Scope of the Project:

- Vendor Module: Vendor registration, product uploads, inventory tracking, and sales monitoring.
- Buyer Module: Product discovery, reviews, order placement, and payment processing.
- Admin Module: Platform monitoring, vendor verification, reporting, and dispute resolution.
- Integration with secure payment gateways and database systems.

## 1.3 Stakeholders

The project involves multiple stakeholders, each with distinct roles and expectations:

- **Vendors:** Individuals or organizations offering products for sale on the platform.
- **Buyers:** End users who browse, purchase, and review products.
- **Administrators:** Responsible for maintaining system integrity, verifying vendors, and ensuring compliance.
- **Technical Team:** Developers, designers, and testers responsible for system development and maintenance.
- **Investors/Owners:** The project initiators who fund and oversee the platform's implementation and growth.

## 1.4 Technology Stack

The proposed *Multi-Vendor E-Commerce Marketplace* leverages a modern and scalable technology stack to ensure high performance, security, and maintainability. The stack has been organized into four categories: **Frontend**, **Backend**, **Development Tools & Dependencies**, and **Design Frameworks**.

### Frontend

- **Next.js 15:** A modern React-based framework for building fast, SEO-optimized, and scalable web applications.
- **React 19.1.0:** A popular JavaScript library for creating interactive user interfaces.
- **Tailwind CSS 4.1.11:** A utility-first CSS framework for rapid and responsive UI development.
- **Shaden UI:** A customizable component library that ensures consistency and efficiency in interface design.
- **DM Sans Font:** A clean, modern, and highly readable font used for enhanced typography.

### Backend

- **Bun Runtime:** An ultra-fast JavaScript runtime environment optimized for speed and performance.
- **Node.js v23.7.0:** A widely adopted server-side runtime enabling scalable backend development.
- **Payload CMS:** A headless CMS providing robust content management and API capabilities for admin operations.
- **MongoDB:** A NoSQL database designed for handling product, user, and transaction data efficiently.

- **Stripe Connect:** A secure and reliable payment gateway for vendor onboarding and financial transactions.

## Development Tools & Dependencies

- **TypeScript 5.8.3:** A strongly typed superset of JavaScript that enhances reliability and maintainability.
- **ESLint 9.31.0:** A tool for code linting and style enforcement to maintain clean and consistent code.
- **Git & GitHub:** Version control and collaborative code management platforms.
- **Shadcn CLI 2.9.2:** A command-line tool for streamlined setup and integration of UI components.

## Design Frameworks

- **NeoBrutalism Theme:** A bold, high-contrast design style providing a modern and distinctive visual appeal.
- **Mobile-First Responsive Layout:** Ensures accessibility and usability across devices of varying screen sizes.



## Chapter 2: Software Requirements Specification & Analysis

### 2.1 Stakeholder Needs & Analysis

The primary and secondary stakeholders for our *Multi-Vendor E-Commerce Marketplace* project are categorized as follows:

#### Primary Stakeholders

Primary stakeholders are individuals or groups directly involved in the core operation and use of the system.

- **Vendors:** Sell products, manage listings, and rely on the platform for business.
- **Customers:** Purchase products, leave reviews, and engage with the marketplace.
- **Admin:** Manages the platform's users, transactions, and overall operations.
- **Software Developers/Designers:** Build and maintain the platform's technical infrastructure.

#### Secondary Stakeholders

Secondary stakeholders are individuals or groups who support the primary stakeholders and influence the platform's success indirectly.

- **Payment Gateway Providers:** Facilitate secure financial transactions.
- **Courier/Delivery Services:** Handle product delivery logistics.
- **Marketing & SEO Team:** Promote the platform and attract users.

#### Requirements Elicitation

Requirements elicitation is the process of gathering requirements from users, customers, and other stakeholders. The objective is to combine problem-solving, elaboration, negotiation, and specification into a structured process.

During this project, several challenges were faced, such as limited communication with stakeholders, time constraints, and volatility of requirements. The following methods were used for requirement elicitation:

- **Surveys:** Questionnaires collected feedback from potential customers and vendors about expectations and pain points in online shopping.
- **Interviews:** One-on-one conversations with vendors, admins, and developers provided deeper insights.
- **Focus Groups:** Small group discussions explored real-world scenarios and gathered honest opinions.

- **Observation:** User interactions with similar platforms were analyzed to identify usability issues and improvement areas.
- **Brainstorming:** Team sessions generated innovative feature ideas.
- **Prototyping:** Mockups and early versions were created to gather user feedback.
- **Use Case/Scenario Analysis:** Real-world scenarios were described to uncover hidden needs and functional requirements.

## 2.2 List of Requirements

### Functional Requirements (FRs)

Functional requirements define the core features and actions of the system to allow users to complete their tasks effectively. For the *Multi-Vendor E-Commerce Marketplace*, these include:

- User authentication for secure registration, login, and logout.
- Vendor store management including product creation, editing, and deletion.
- Product browsing, search, and filtering.
- Payment method options (online payment and cash on delivery).
- Shopping cart functionality for adding, updating, and removing items.
- Order management system for confirmation or cancellation.
- Role-based access control for customers, vendors, and admins.
- Product review and rating system for customer feedback.
- Admin dashboard for user management, transaction monitoring, and content moderation.
- Contact and notification system (emails, SMS, and helpline support).

### Non-Functional Requirements (NFRs)

Non-functional requirements define how the system should perform to ensure quality, reliability, and usability. For this project, they include:

- **Security:** Protect user data and prevent cyber threats.
- **Performance:** Ensure fast loading times and smooth operations.
- **Scalability:** Support growth in users, products, and transactions.
- **Availability:** Maintain 24/7 uptime with minimal downtime.
- **Usability:** Provide an intuitive and user-friendly interface.
- **Data Integrity:** Ensure accuracy and consistency of information.
- **Accessibility:** Support access from any internet-connected device.

## Extra-Ordinary Requirements

Extra-ordinary requirements go beyond the standard functional and non-functional expectations, adding unique value:

- Advanced product browsing and filtering (by name, category, price, discount, rating, and availability).
- Order management system with history tracking and real-time delivery updates.
- Wishlist and favorites feature for saving products for later purchase.

## 2.3 House of Quality (QFD Integration)

### Customer Requirements (CRs)

- **Easy Product Search and Filtering:** Intuitive search bar with category filters, price range, ratings, and vendor-based filtering.
- **Secure Payment Process:** Encrypted transactions with multiple gateways (Stripe, PayPal) and PCI-DSS compliance.
- **Fast Delivery Options:** Integration with logistics APIs to display real-time shipping estimates and multiple delivery speeds.
- **Vendor Profile and Reviews:** Dedicated vendor profiles with ratings, contact details, and product listings to build trust.
- **Mobile Responsiveness:** Fully responsive design for smartphones and tablets.
- **Customer Support Availability:** Live chat and chatbot integration for 24/7 support.
- **Easy Return and Refund Process:** Transparent return/refund system with vendor-specific policies.

### Engineering Requirements (TRs)

- **Advanced Search Algorithm:** Implementation of Elasticsearch/Algolia for fast and scalable searches.
- **Integration with Stripe/PayPal:** Secure handling of transactions, vendor payments, and fraud detection.
- **Logistics API Integration:** Real-time shipping data via APIs (FedEx, DHL, Ship Rocket).
- **Vendor Management Module:** Vendor dashboards with inventory, pricing, and order tracking.
- **Responsive Web Design:** Mobile-first design using Tailwind CSS or Bootstrap.
- **Return Management System:** Automated return center with vendor and admin monitoring, supporting refunds and exchanges.

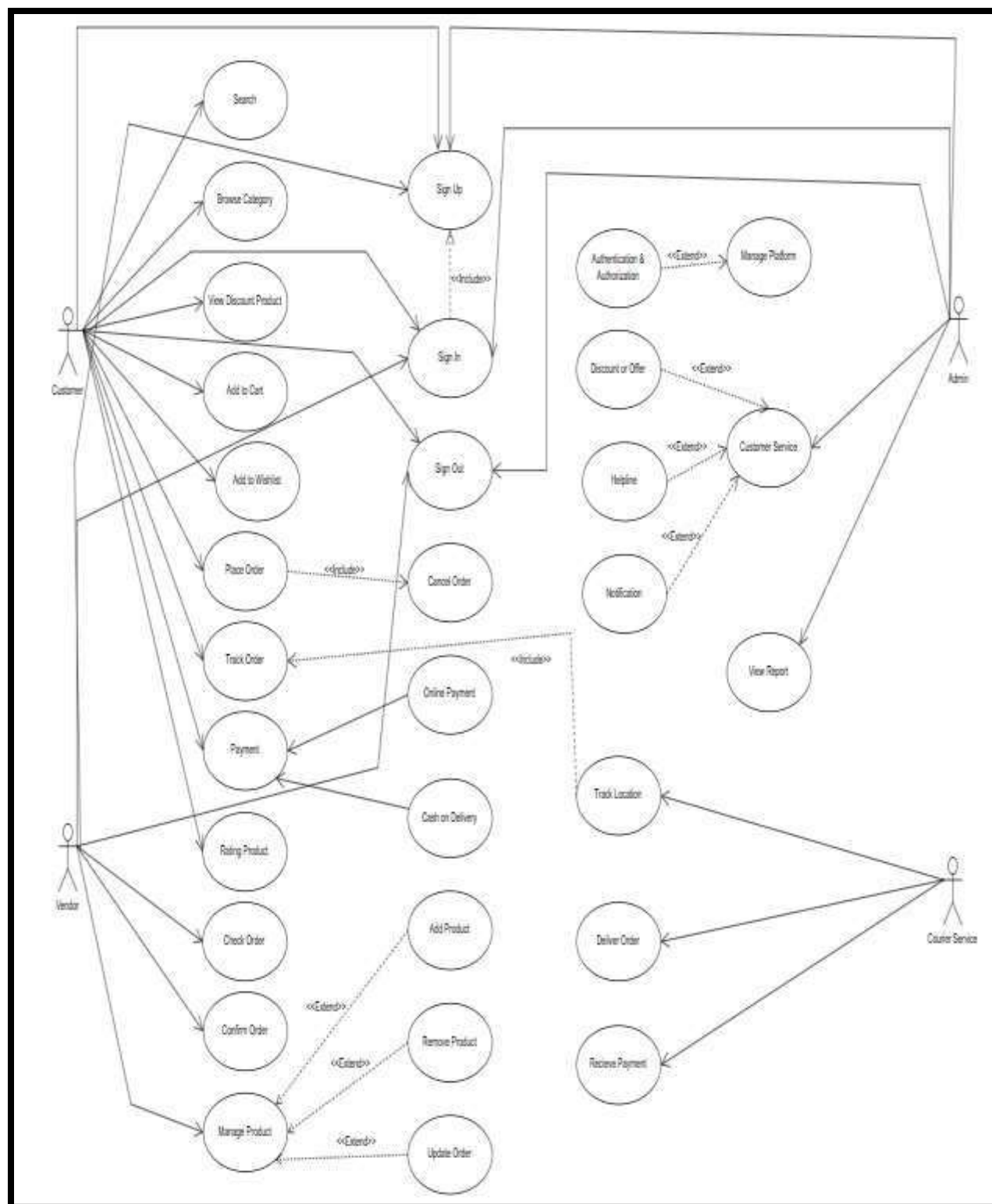
**Table 1.1:** Quality Function Deployment (QFD) Matrix for Bazario

Customer Requirements (CRs)	Search Algo	Payment Integration	Logistics API	Vendor Module	Chat System	Chat System	Return System
1. Easy search/filtering	Strong				Weak		
2. Secure payment		Strong					
3. Fast delivery options			Strong				
4. Vendor profile/reviews				Strong			
5. Mobile responsiveness	Weak	Weak	Weak	Weak	Strong	Weak	Weak
6. Customer support availability						Strong	
7. Easy return/refund process							Strong

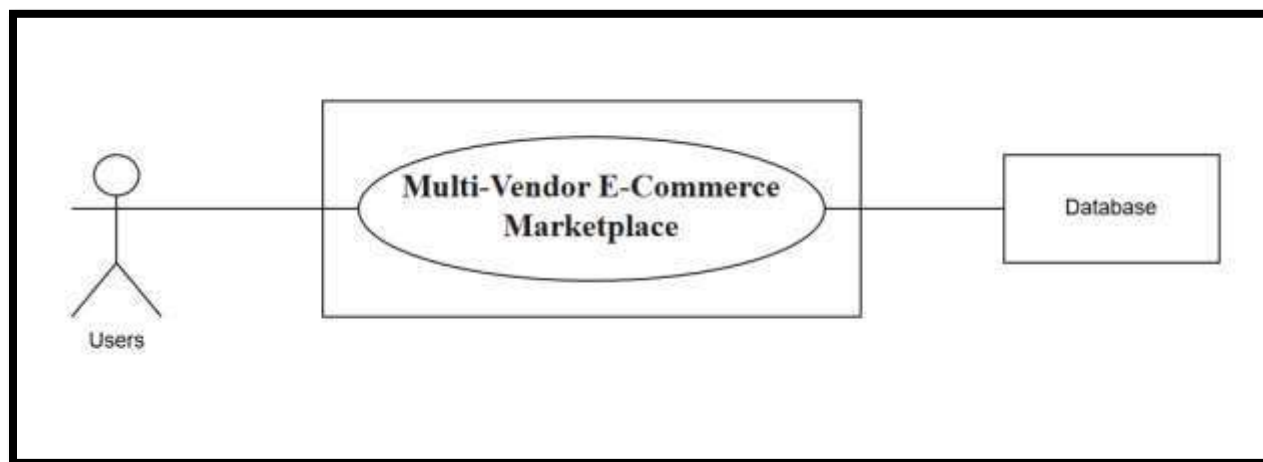
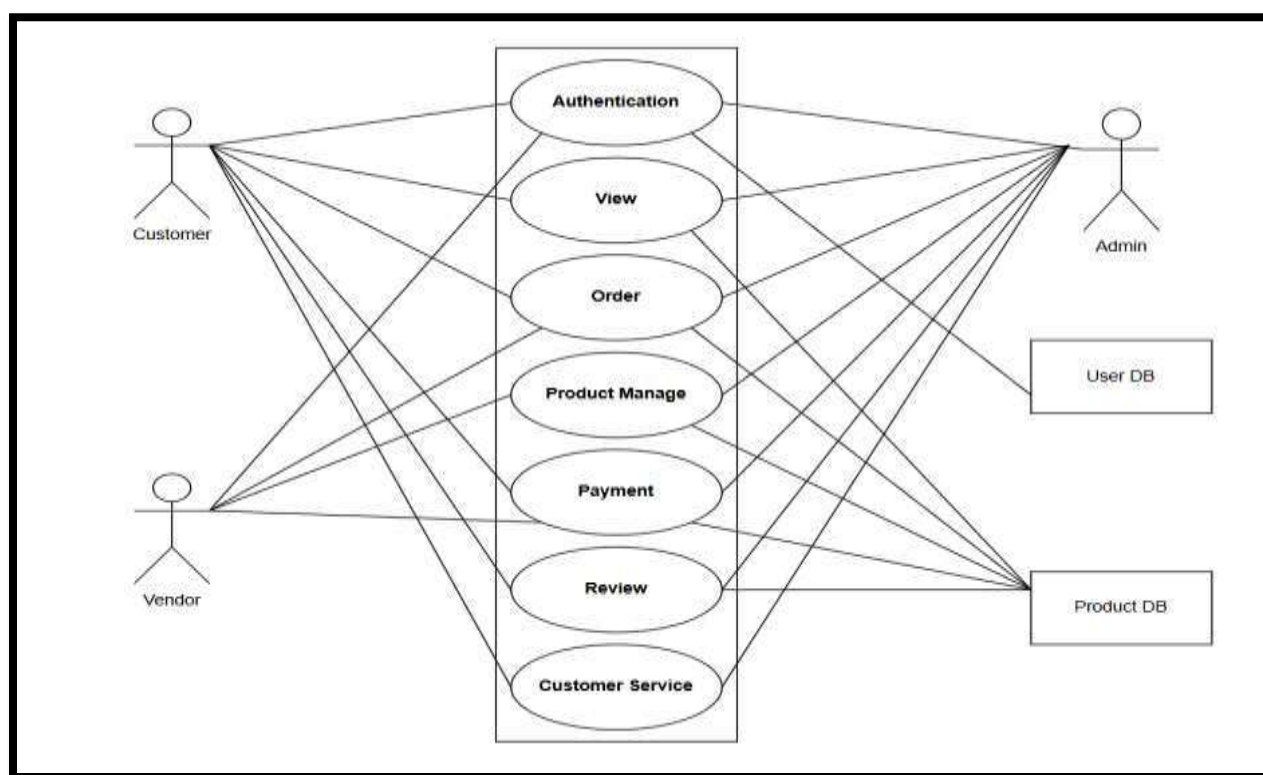
## 2.4 Requirements Modeling

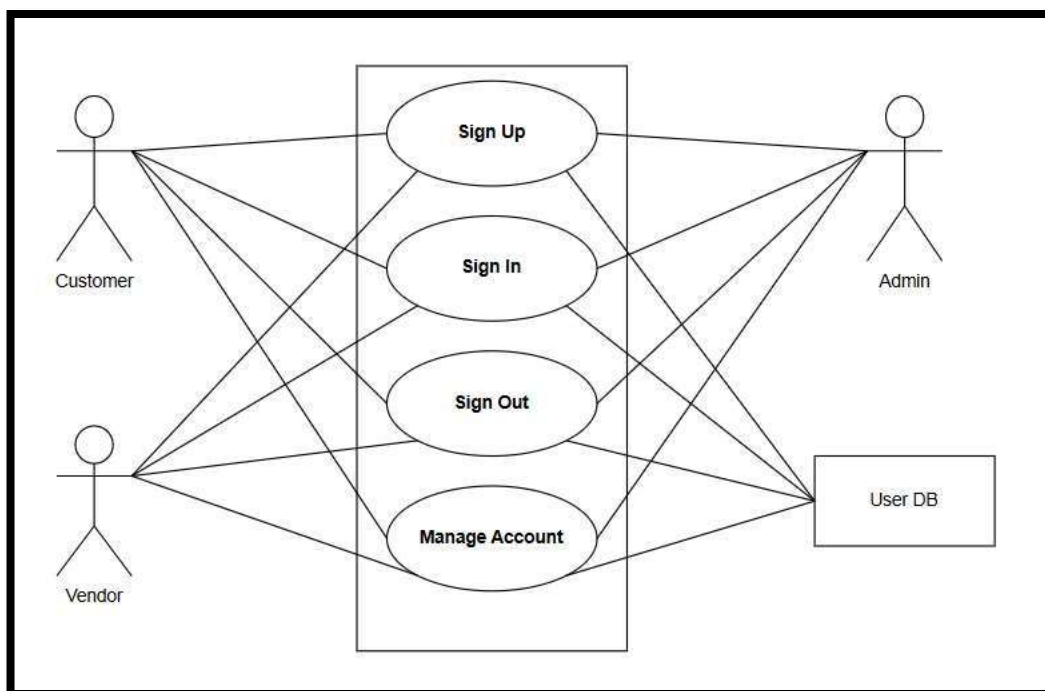
### 2.4.1 Use Case Diagrams

Use Case Diagrams provide a high-level visualization of how different stakeholders interact with the system. For this project, they illustrate interactions among customers, vendors, and admins with features such as product browsing, vendor store management, payment processing, and order tracking. These diagrams help identify system boundaries and clarify functional requirements.

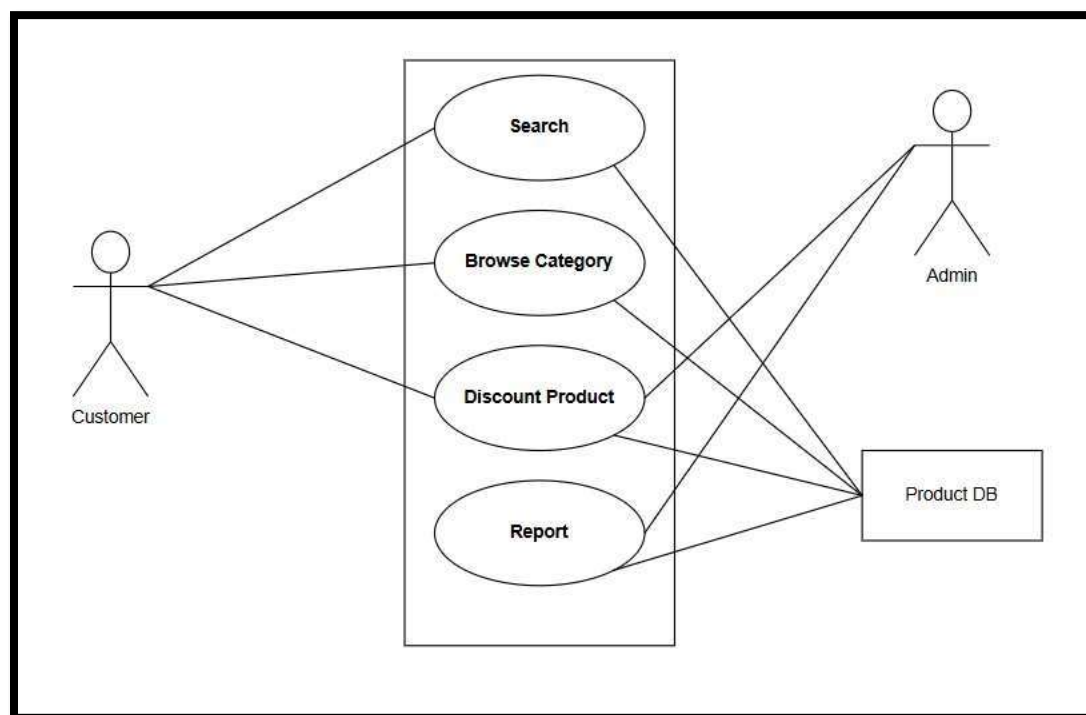


**Figure 1.1:** Use Case Diagram for Bazarrio System

**Use case diagram Step by Step:****Figure 1.2:** Level 0 for Multi-Vendor E-Commerce Marketplace**Figure 1.3:** Level 1 for Multi-Vendor E-Commerce Marketplace

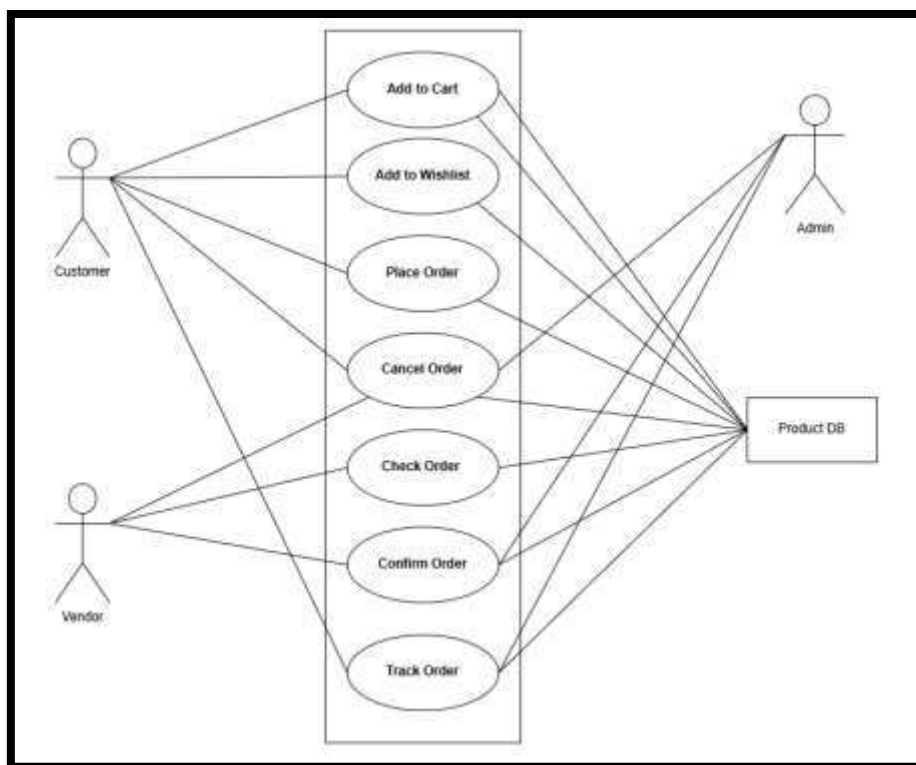


**Figure 1.4:** Level 2.1 for Multi-Vendor E-Commerce Marketplace

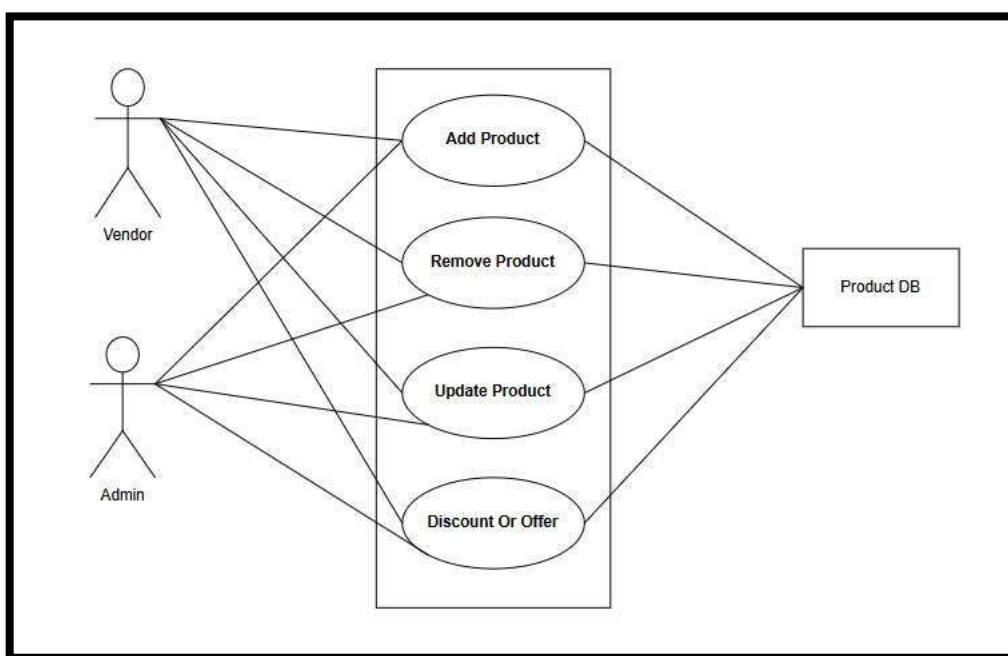


**Figure 1.5:** Level 2.2 for Multi-Vendor E-Commerce Marketplace



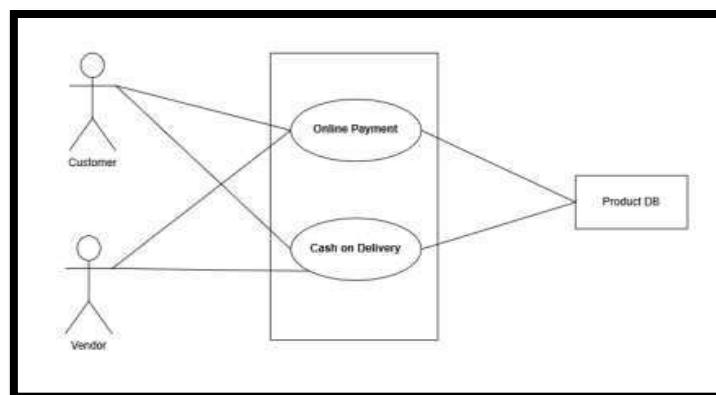


**Figure 1.6:** Level 2.3 for Multi-Vendor E-Commerce Marketplace

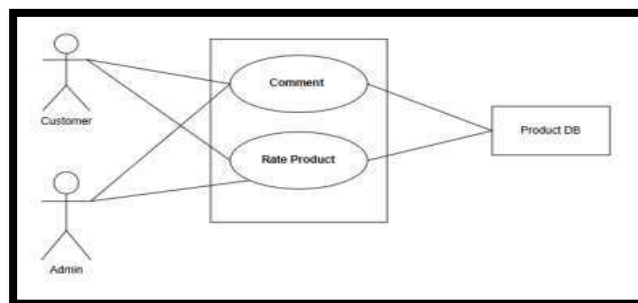


**Figure 1.7:** Level 2.4 for Multi-Vendor E-Commerce Marketplace

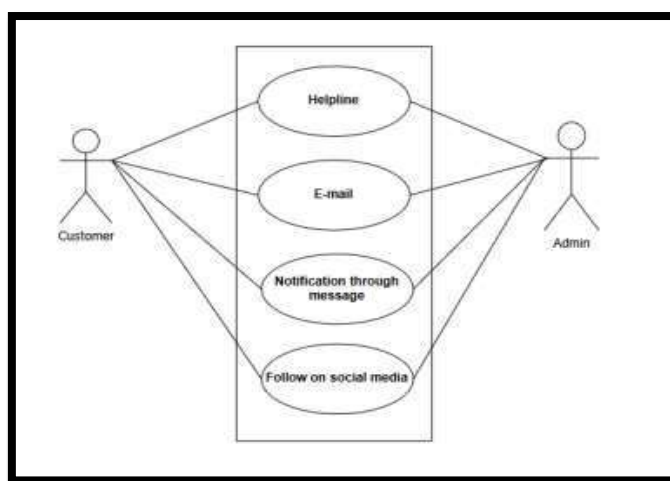




**Figure 1.8:** Level 2.5 for Multi-Vendor E-Commerce Marketplace



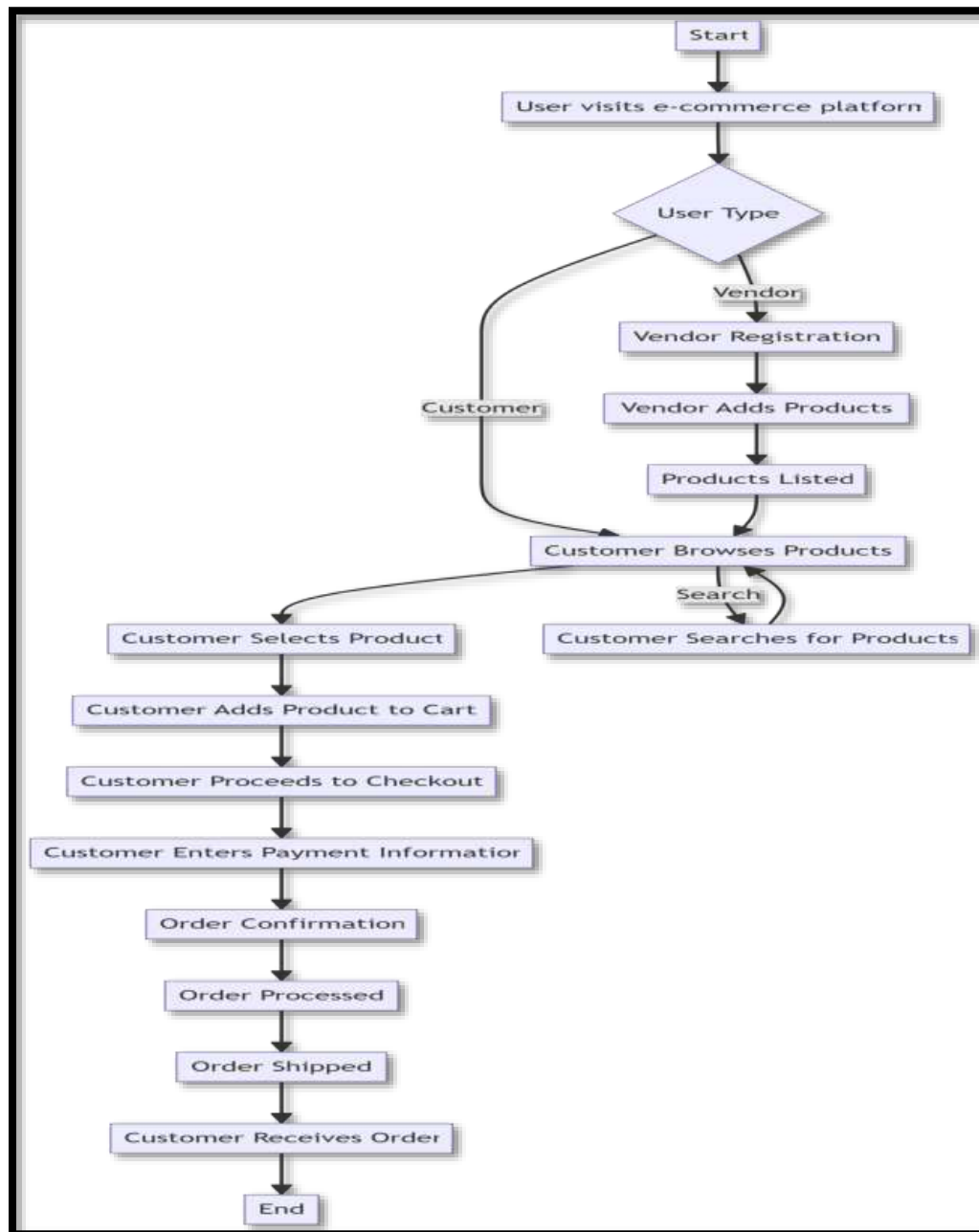
**Figure 1.9:** Level 2.6 for Multi-Vendor E-Commerce Marketplace



**Figure 1.10:** Level 2.7 for Multi-Vendor E-Commerce Marketplace

## 2.4.2 Activity Diagrams

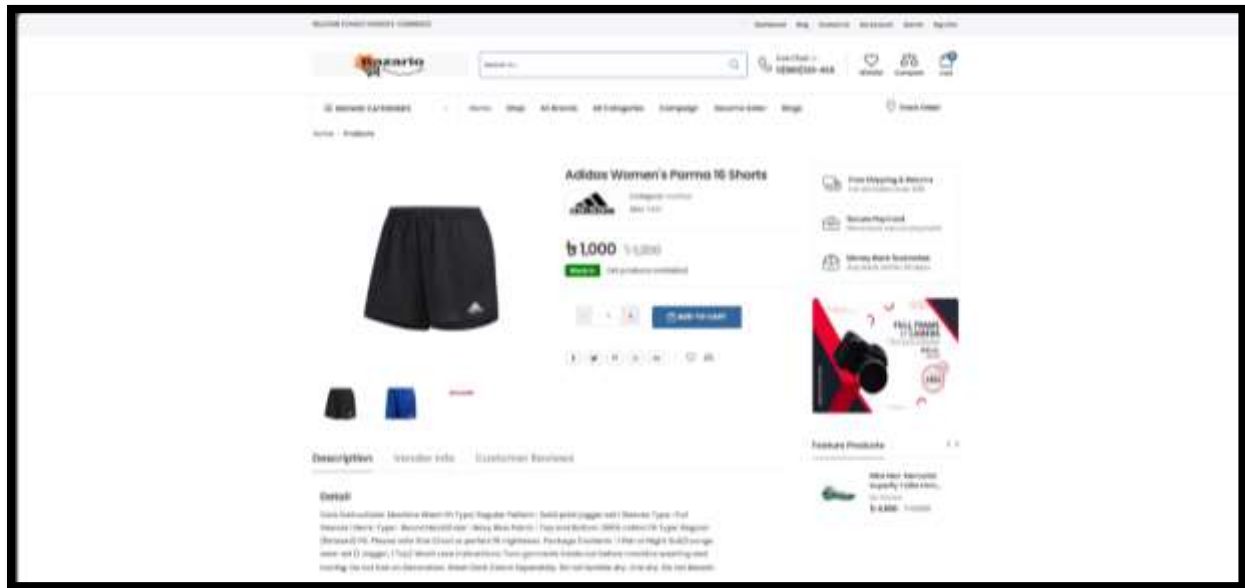
Activity Diagrams model the flow of activities within the system. They describe how users perform tasks such as customer checkout, vendor product management, and admin approval workflows.



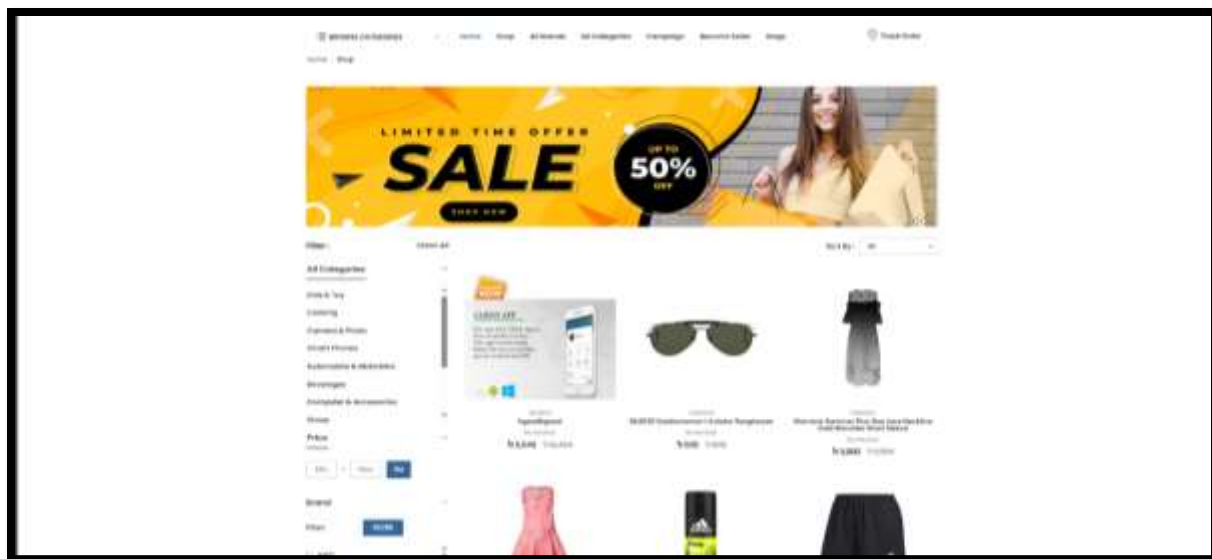
**Figure 2:** Activity Diagram for Multi-Vendor E-Commerce Marketplace

### 2.4.3 UI Sketches

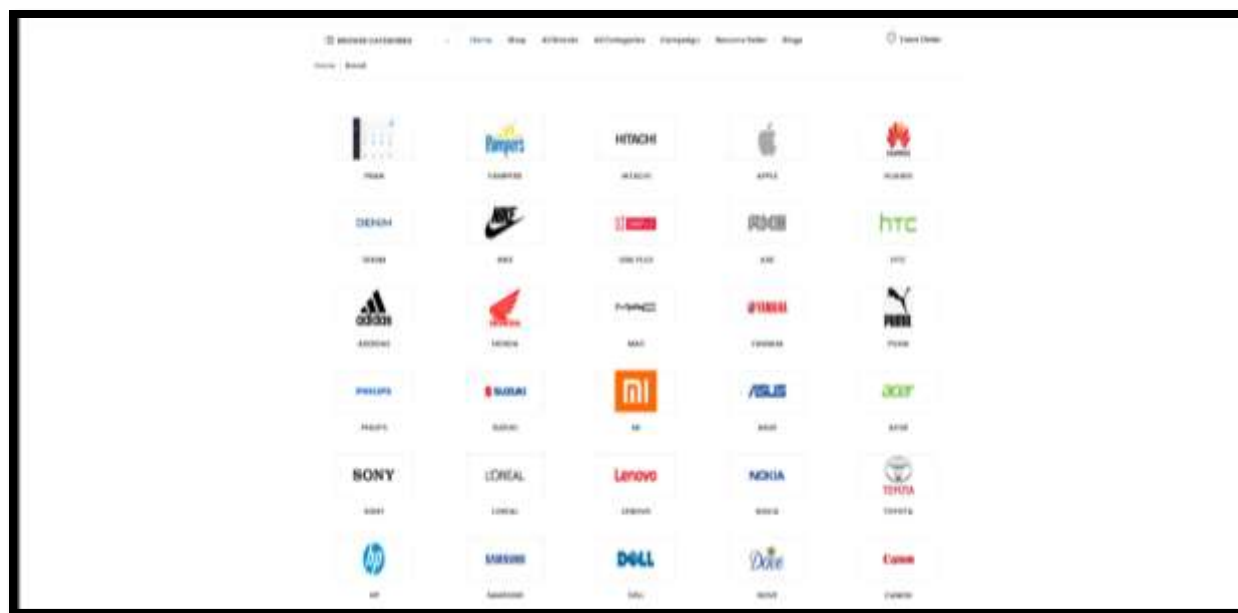
UI Sketches present the initial design concepts of the platform's user interface. They provide a visual layout of key pages such as the homepage, product detail pages, shopping cart, vendor dashboard, and admin panel. These sketches focus on usability and navigation flow, ensuring that users can interact with the platform intuitively.



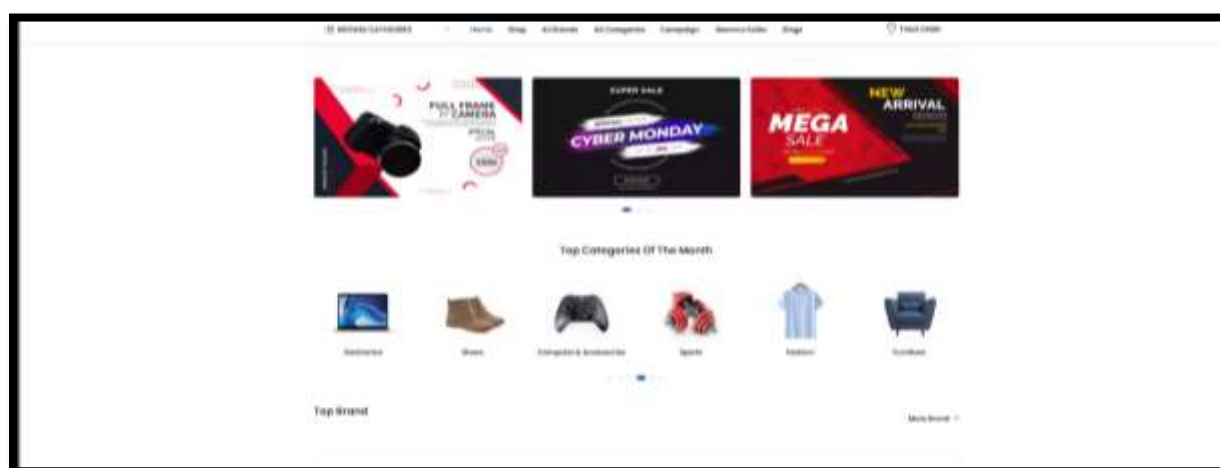
**Figure 3.1:** Products Interface for Users in Multi-Vendor E-Commerce Marketplace



**Figure 3.2:** Products Interface for Users in Multi-Vendor E-Commerce Marketplace



**Figure 3.3:** Products Interface for Users in Multi-Vendor E-Commerce Marketplace



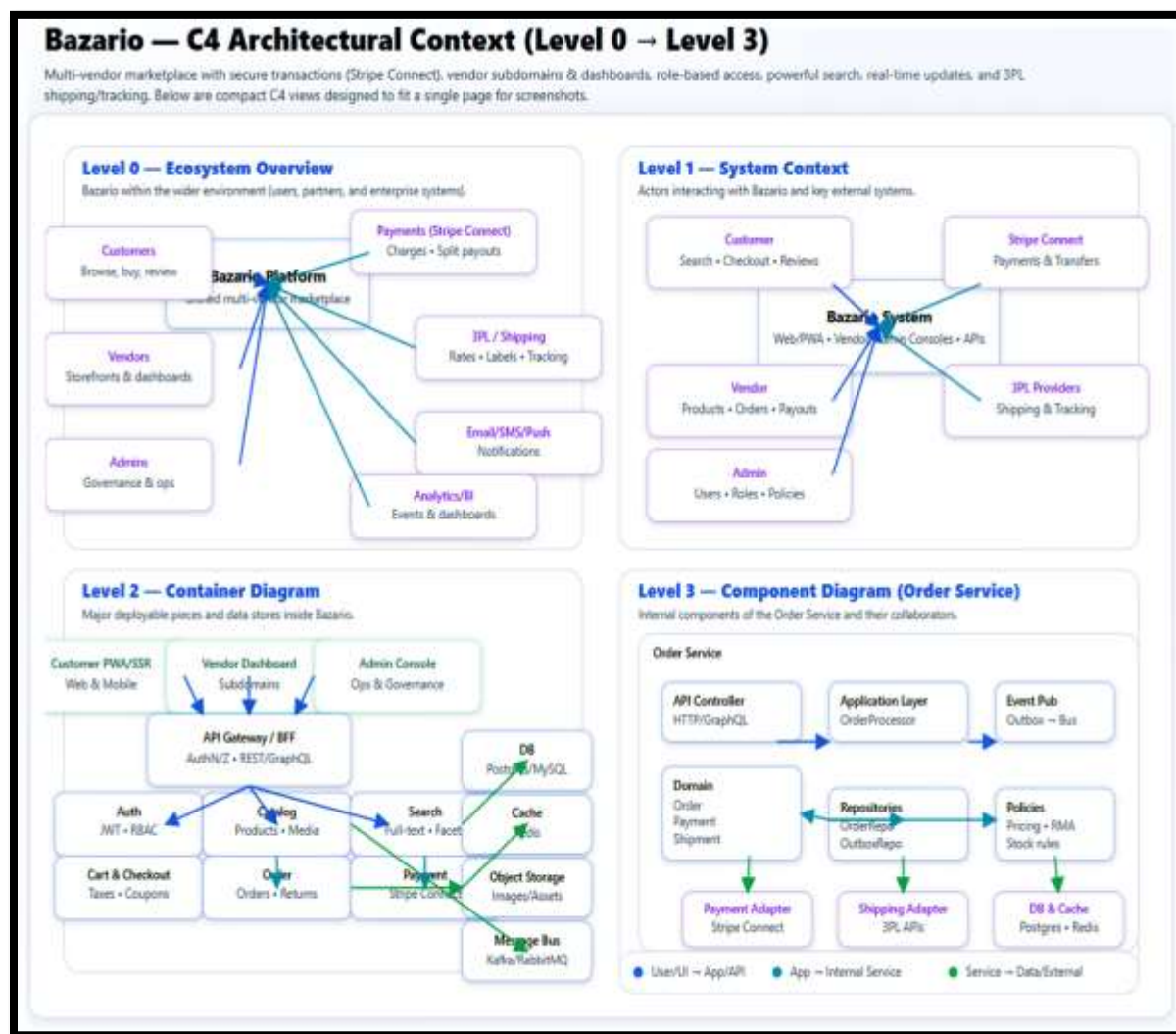
**Figure 3.4:** Products Interface for Users in Multi-Vendor E-Commerce Marketplace

## Chapter 3: Software Design

### 3.1 Architectural Design

#### 3.1.1 Architectural Context Diagram

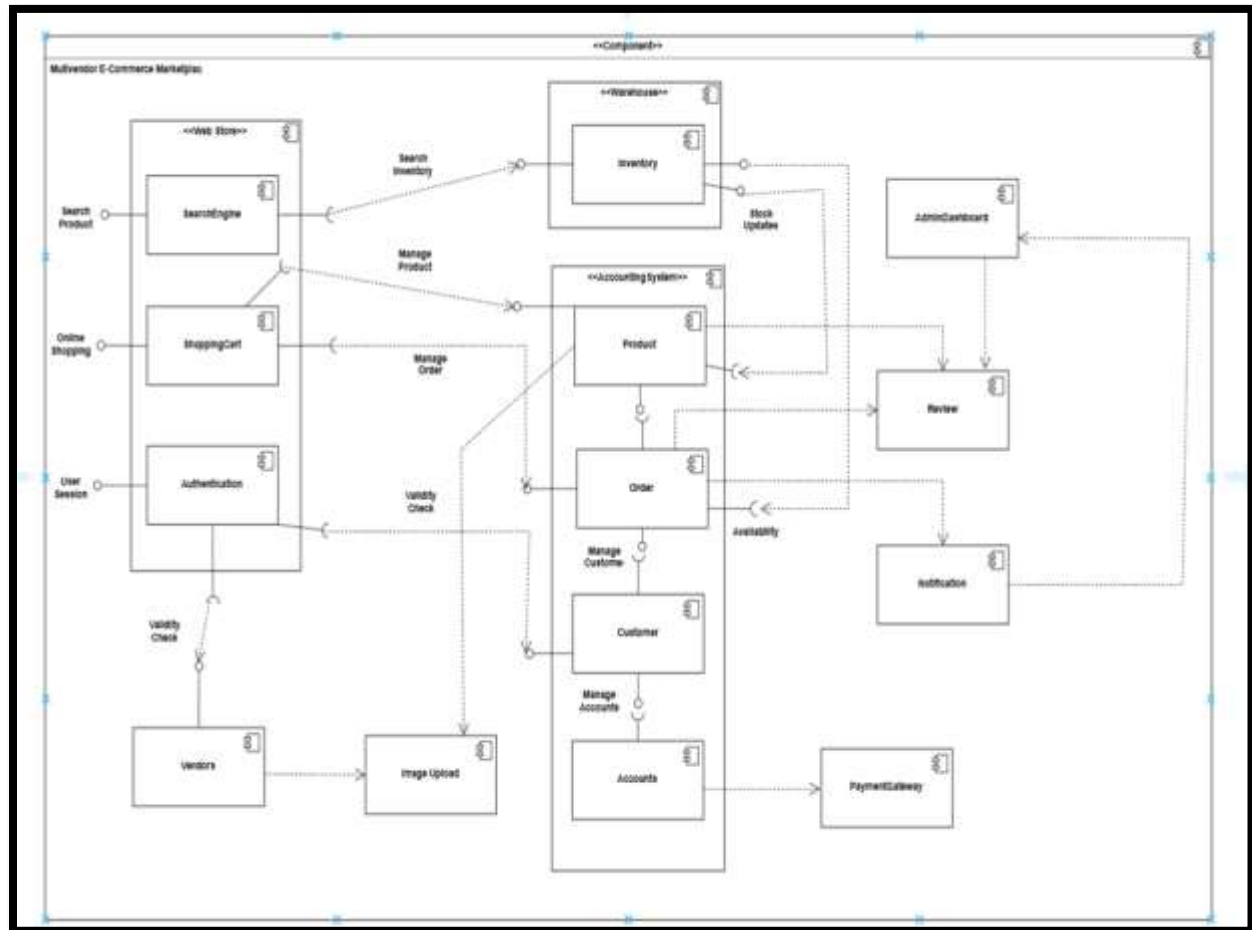
The architectural context diagram illustrates the high-level interactions between the *Multi-Vendor E-Commerce Marketplace* and external entities such as vendors, customers, payment gateways, courier services, and administrators. It highlights the data flow, system boundaries, and external dependencies, ensuring a clear understanding of the system's scope.



**Figure 4:** Architectural Context Diagram of the Multi-Vendor Marketplace

### 3.1.2 Top-Level Component Diagram

The top-level component diagram decomposes the system into major components including the user interface, vendor module, customer module, admin module, payment services, and logistics integration. This diagram shows how components communicate and depend on each other to deliver system functionality.



**Figure 5:** Top-Level Component Diagram of the System

## 3.2 Component-Level Design

### 3.2.1 Elaboration of Design Components

At the component level, the design specifies the structure and behavior of each module in detail. This includes module interfaces, input/output handling, and internal workflows.

#### 1. Web-Store Subsystem

- **Search Engine**
  - *Instantiation:* Manages product searches within the online store.
  - *Elaboration:*
    - Performs keyword, category, and filter-based searches.
    - Retrieves stock data from Inventory.
    - Returns ranked and paginated product lists.
- **Shopping Cart**
  - *Instantiation:* Provides temporary storage for items users intend to purchase.
  - *Elaboration:*
    - Adds, removes, and updates items.
    - Calculates subtotals, discounts, and taxes.
    - Passes confirmed items to Orders for processing.
- **Authentication**
  - *Instantiation:* Securely manages user identity and access.
  - *Elaboration:*
    - Handles registration, login, and logout.
    - Maintains active sessions during shopping.
    - Works with Customers and Vendors for profile data access.

#### 2. Warehouses Subsystem

- **Inventory**
  - *Instantiation:* Tracks stock levels for all products.
  - *Elaboration:*
    - Updates quantities after purchases or restocking.
    - Provides availability data to Search Engine and Orders.
    - Manages inventory locations and batch tracking.

### 3. Accounting Subsystem

- **Orders**
  - *Instantiation:* Handles all order creation and management processes.
  - *Elaboration:*
    - Receives order details from Shopping Cart.
    - Verifies stock availability via Inventory.
    - Tracks order status from pending to delivered.
- **Product Component**
  - *Instantiation:* Central entity representing each item available in the marketplace.
  - *Elaboration:*
    - Stores product data: name, description, SKU, category, brand, price, and stock unit.
    - Links with Image Upload for product images.
    - Interacts with Inventory for stock updates.
    - Connects to Review for displaying ratings and feedback.
    - Integrates with Shopping Cart and Orders to track sales.
    - Associates each product with a vendor's profile.
    - Handles product variations such as size, color, and attributes.
- **Customers**
  - *Instantiation:* Manages customer data and relationships.
  - *Elaboration:*
    - Stores addresses, contact information, and preferences.
    - Links with Authentication for account validation.
    - Provides order history for analytics.
- **Accounts**
  - *Instantiation:* Manages all financial transactions.
  - *Elaboration:*
    - Records payments, refunds, and payouts.
    - Integrates with Payment Gateway.
    - Generates financial reports for admins and vendors.

### 4. Added Components

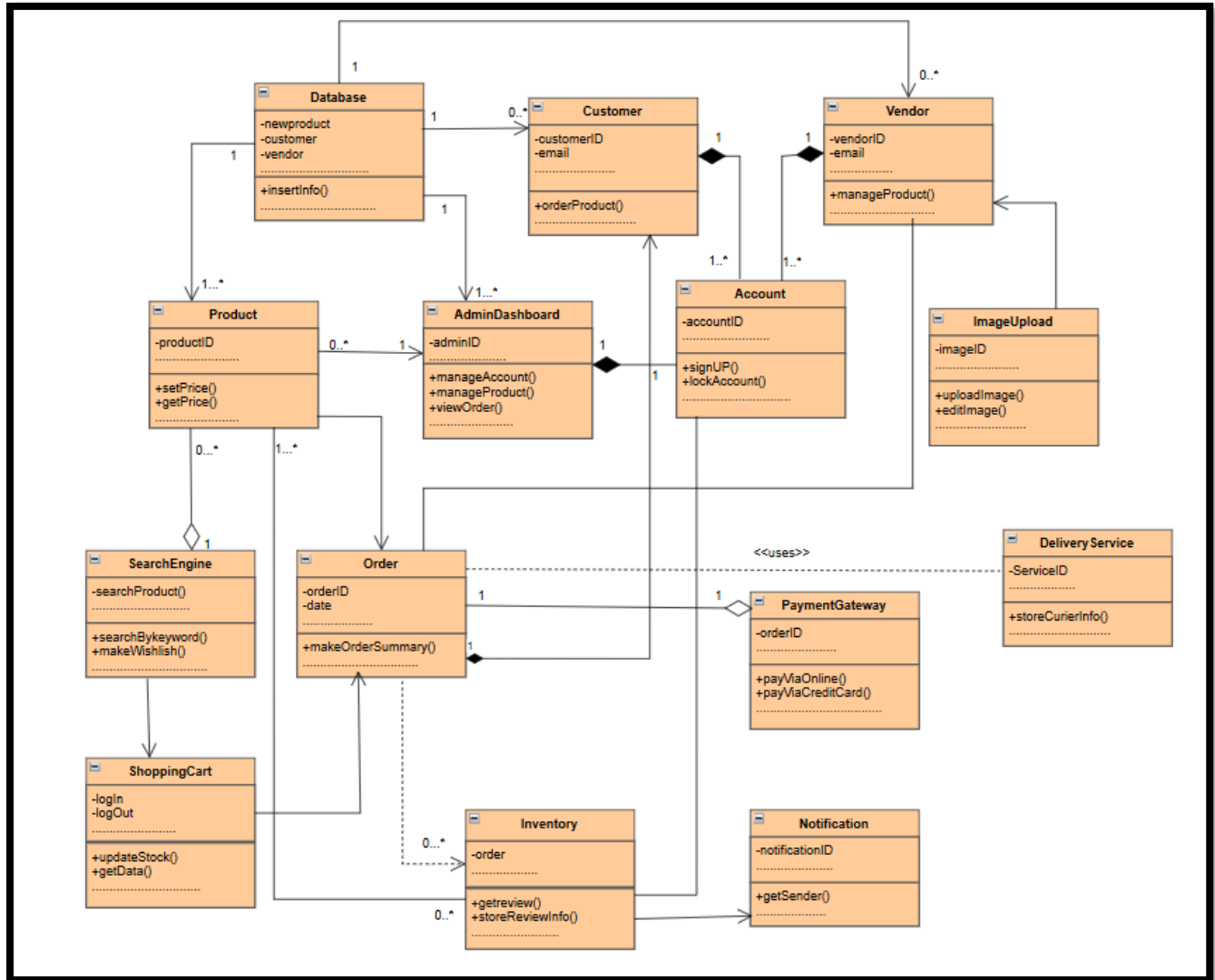
- **Payment Gateway**
  - *Instantiation:* Processes online payments securely.
  - *Elaboration:*
    - Integrates with Accounts to verify transactions.
    - Supports multiple payment methods (credit card, wallet, bank transfer).
    - Ensures encryption and fraud detection.
- **Review**
  - *Instantiation:* Enables customer feedback for products.
  - *Elaboration:*
    - Allows customers to submit ratings and reviews.



- Links reviews to specific Orders and Products.
  - Provides moderation tools for admins.
- **Notification**
  - *Instantiation:* Sends system alerts and updates to users.
  - *Elaboration:*
    - Informs customers about order status changes.
    - Sends promotional and security notifications.
    - Works with Orders and Admin Dashboard for timely updates.
- **Admin Dashboard**
  - *Instantiation:* Central control panel for administrators.
  - *Elaboration:*
    - Manages vendors, products, orders, and customers.
    - Generates reports on sales, revenue, and system health.
    - Provides moderation for Reviews and content.
- **Image Upload**
  - *Instantiation:* Manages product and profile image storage.
  - *Elaboration:*
    - Supports multiple formats and sizes.
    - Optimizes images for performance (thumbnails, compression).
    - Integrates with vendor product listing tools.
- **Vendor**
  - *Instantiation:* Provides the vendor-side management interface.
  - *Elaboration:*
    - Allows vendors to add, edit, and remove products.
    - Enables vendors to view and fulfill orders.
    - Provides analytics on sales and performance.

### 3.2.2 Class Diagram

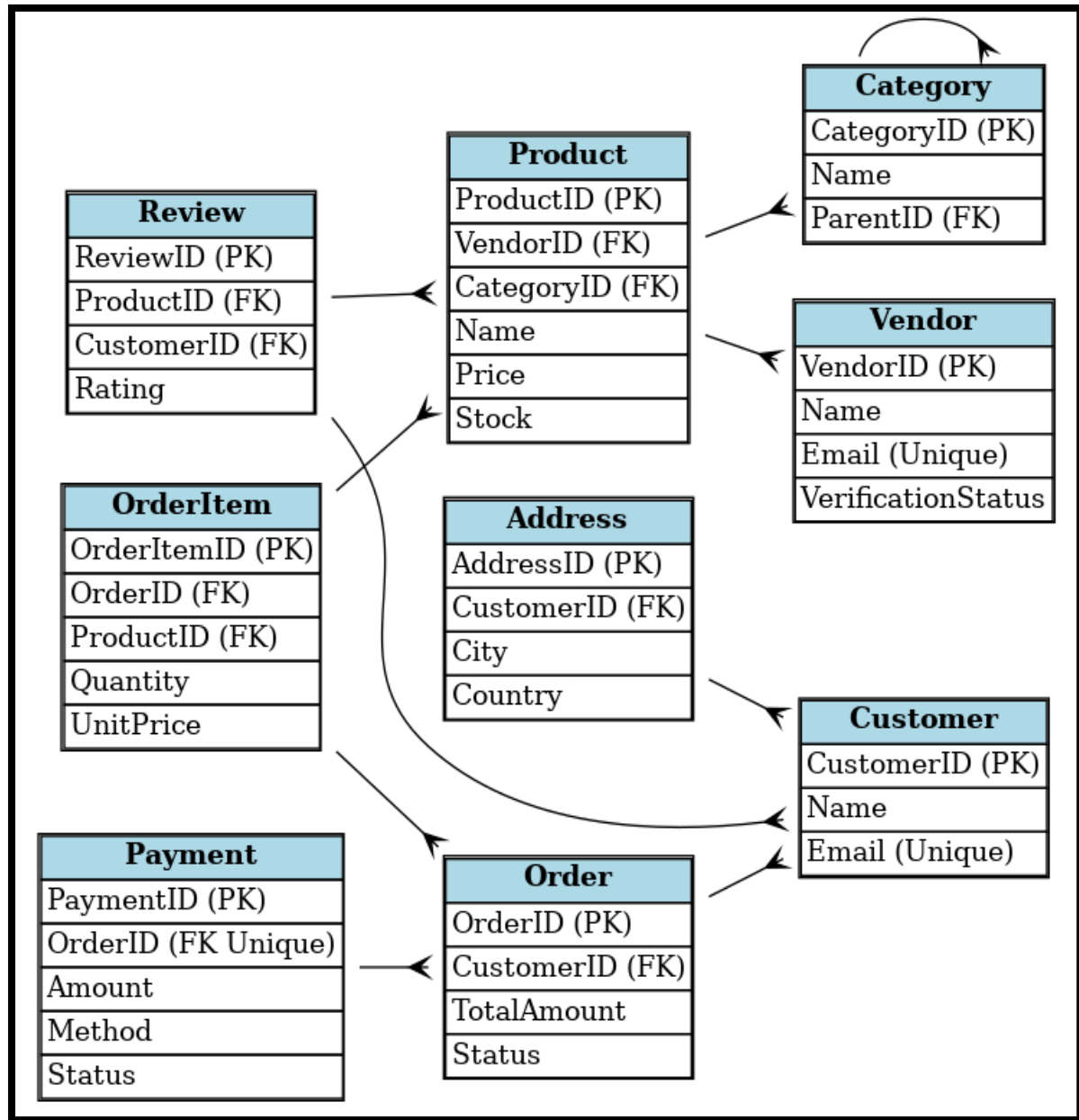
The class diagram illustrates the object-oriented structure of the system, including classes, attributes, methods, and relationships (associations, inheritance, and dependencies). It provides a blueprint for implementation by showing how system entities such as *User*, *Vendor*, *Customer*, *Product*, *Order*, and *Payment* interact.



**Figure 6:** Class Diagram of Core Entities

### 3.2.3 Database Design

The database design defines how system data is stored, managed, and retrieved. It includes an **Entity-Relationship (ER) Diagram** and relational schema to represent key entities such as *Users*, *Products*, *Orders*, *Transactions*, and *Reviews*. This design ensures data integrity, scalability, and efficient query performance.



**Figure 7:** Database ER Diagram of the Marketplace

## 3.3 User Interface Design

### 3.3.1 UI Wireframes/Mockups/Prototypes

Wireframes and mockups represent the visual structure of the platform's main pages. These include the homepage, product listing page, product detail page, shopping cart, vendor dashboard, and admin panel. The prototypes emphasize usability, consistency, and modern design principles.



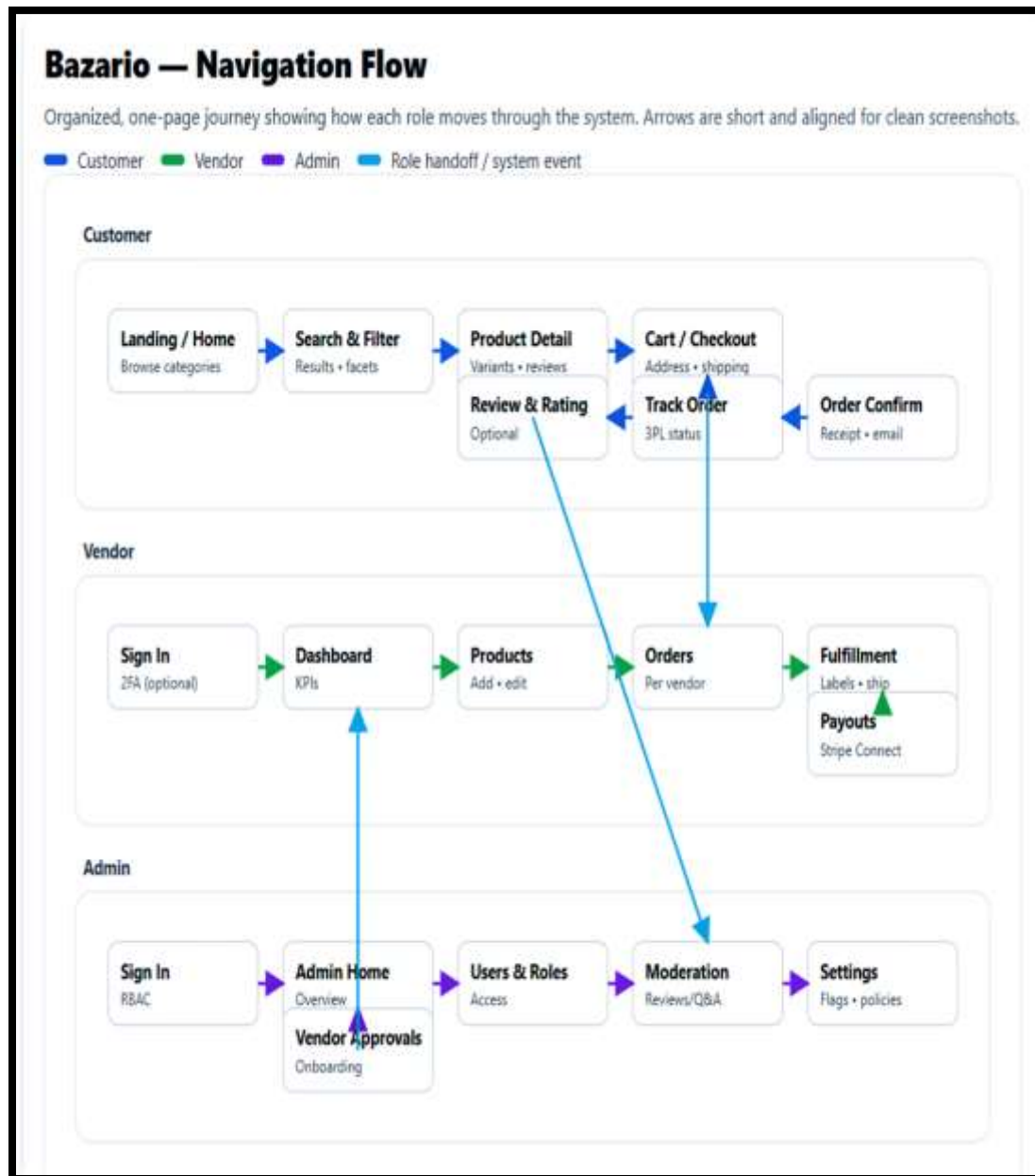
**Figure 8.1:** Wireframe of the Homepage



**Figure 8.2:** Admin Dashboard

### 3.3.2 Navigation Flow

The navigation flow diagram shows how users move between different sections of the application, ensuring intuitive access to key features. It maps the journey from user authentication to browsing, ordering, payment, and post-purchase interactions such as reviews and returns.



**Figure 9.1:** Navigation Flow of the System

## Chapter 4: Implementation

### 4.1 Code Structure Overview

The implementation of the *Multi-Vendor E-Commerce Marketplace* follows a modular structure to ensure maintainability, scalability, and ease of collaboration. The project is divided into well-defined layers for frontend, backend, and database interactions, along with supporting configurations.

#### Frontend (/frontend)

The **frontend** directory contains all files related to the user interface, built with **Next.js** and **React**. It is responsible for rendering pages, handling client-side logic, and providing an interactive experience for customers, vendors, and admins.

- **/components** – Contains reusable React components such as navigation bars, buttons, forms, modals, and product cards. These ensure consistency across the application and improve development efficiency.
- **/pages** – Defines application routes like *Home*, *Product Listing*, *Product Details*, *Cart*, *Checkout*, *Vendor Dashboard*, and *Admin Panel*. Next.js automatically maps these files to URLs.
- **/styles** – Holds styling files for the user interface. It primarily uses **Tailwind CSS** for rapid and responsive design customization, along with global theme configuration.
- **/public** – Stores static assets such as fonts, icons, images, and other resources that are directly accessible by the browser.

#### Backend (/backend)

The **backend** directory implements the server-side logic using **Node.js** and **Payload CMS**. It ensures secure data management, API handling, and communication between the database and the frontend.

- **/controllers** – Manages incoming HTTP requests, applies business logic, and communicates with models. For example, the `OrderController` handles placing and tracking orders.
- **/models** – Contains **MongoDB schemas** representing entities such as *Product*, *User*, *Order*, and *Review*. These define the structure of stored data.
- **/routes** – Defines REST API endpoints for vendors, customers, and administrators. For instance, `/api/products` retrieves product listings.
- **/services** – Contains reusable business logic such as authentication, payment integration, notifications, and vendor analytics. These services keep controllers lightweight.
- **/config** – Stores environment variables, database connection settings, and middleware configuration for security, validation, and logging.

### Database (/database)

This folder contains scripts and configurations for managing the **MongoDB database**. It includes migration scripts for updating database schema and ensures data integrity during development and deployment.

### Design (/design)

The **design** folder holds UI/UX artifacts including **wireframes, mockups, and prototypes**. It documents the visual planning phase and provides references for frontend developers.

### Tests (/tests)

This directory includes **unit tests, integration tests, and automated test cases**. These tests validate that components, modules, and APIs work correctly and ensure system reliability before deployment.

### Documentation (/docs)

The **docs** folder stores project-related documents such as the **Software Requirements Specification (SRS), design reports**, and testing reports. It acts as a central reference for both technical and non-technical stakeholders.

### Package.json

The **package.json** file maintains project dependencies, scripts, and metadata. It is critical for managing libraries, running build processes, and ensuring consistent setup across developer environments.

## 4.2 GitHub Repository URL

The source code and related artifacts of the project are maintained in a **GitHub repository** for collaboration, version control, and deployment.

- **GitHub Repository:**

Backend URL: <https://github.com/SifatSwapnil2022/bazario.com.backend>

Frontend URL: <https://github.com/SifatSwapnil2022/bazario.com>

Website Homepage URL: <https://github.com/SifatSwapnil2022/multitenant-ecommerce>

Deployed URL: <https://bazario.cloudmindsoftware.com/>

The repository contains:

- Source code for frontend and backend.
- Database schema and migration scripts.
- Documentation (SRS, design diagrams, testing reports).
- Test cases and automation scripts.
- CI/CD configurations for deployment.



## Chapter 5: Software Testing

### 5.1 White Box Testing (Code-Level)

White box testing was conducted to validate the **internal logic, control flow, and decision branches** of the system. The focus was on three critical modules: **Login Page, Register Page, and Confirm Order Page**. Each module was analyzed using **cyclomatic complexity** and **decision-to-decision (DD) graphs** to ensure all independent paths were covered.

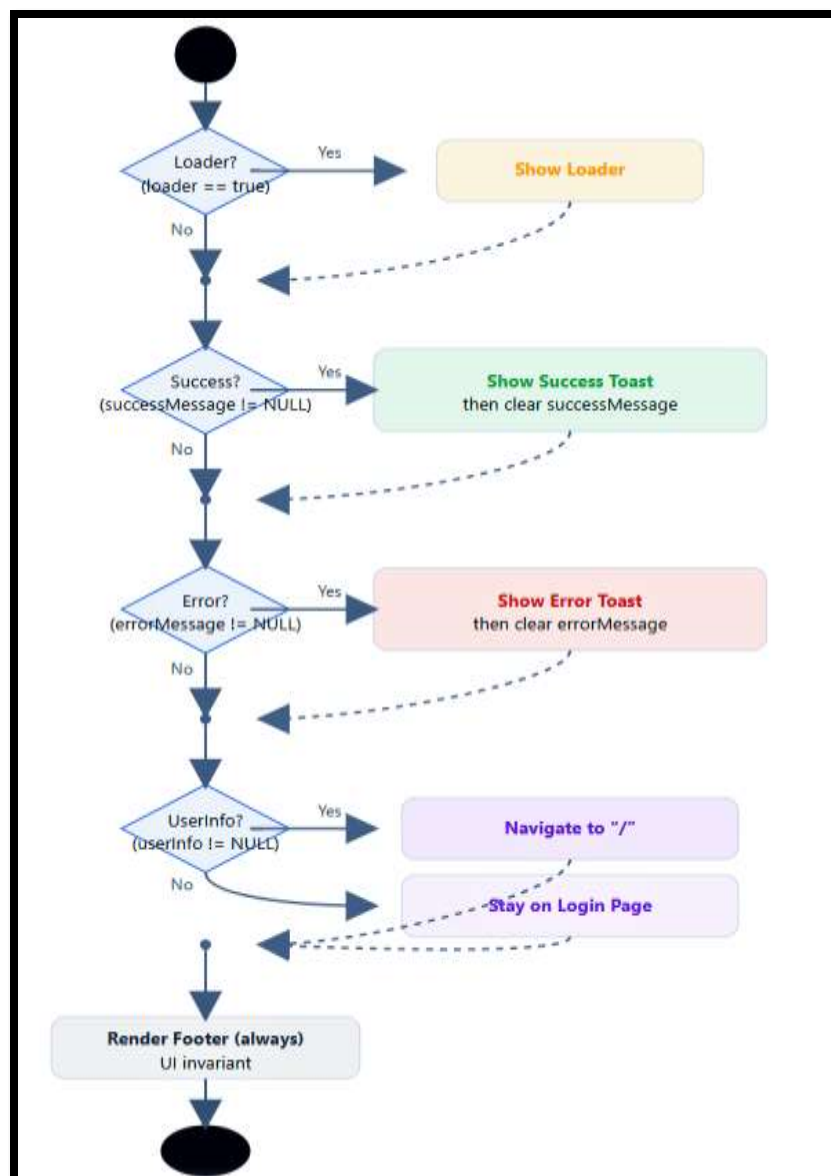
#### 5.1.1 Login Page

The Login Page implements conditional rendering and navigation based on loader states, success messages, error messages, and user session data.

- **Cyclomatic Complexity (CC): 5**
- **Independent Paths:**
  1. Loader → Success → Error → UserInfo → Navigate → Footer → End
  2. Loader → Success → Error → No UserInfo → Stay → Footer → End
  3. Loader → Success → No Error → UserInfo → Navigate → Footer → End
  4. Loader → Success → No Error → No UserInfo → Stay → Footer → End
  5. Loader → No Success → Error → UserInfo → Navigate → Footer → End

Traceability Matrix		
Requirement	Graph Node(s)	Covered By Paths
Show loader if <code>loader == true</code>	Loader?	P1-P5
Success toast & clear if <code>successMessage != NULL</code>	Success?	P1-P4
Error toast & clear if <code>errorMessage != NULL</code>	Error?	P1, P2, P5
Navigate if <code>userInfo != NULL</code> else stay	UserInfo?	P1, P3, P5 (Navigate) • P2, P4 (Stay)
Always render footer	Footer	All paths

**Figure 10.1:** Paths Covered for Login Module



**Figure 10.2:** Decision-to-Decision Graph for Login Module

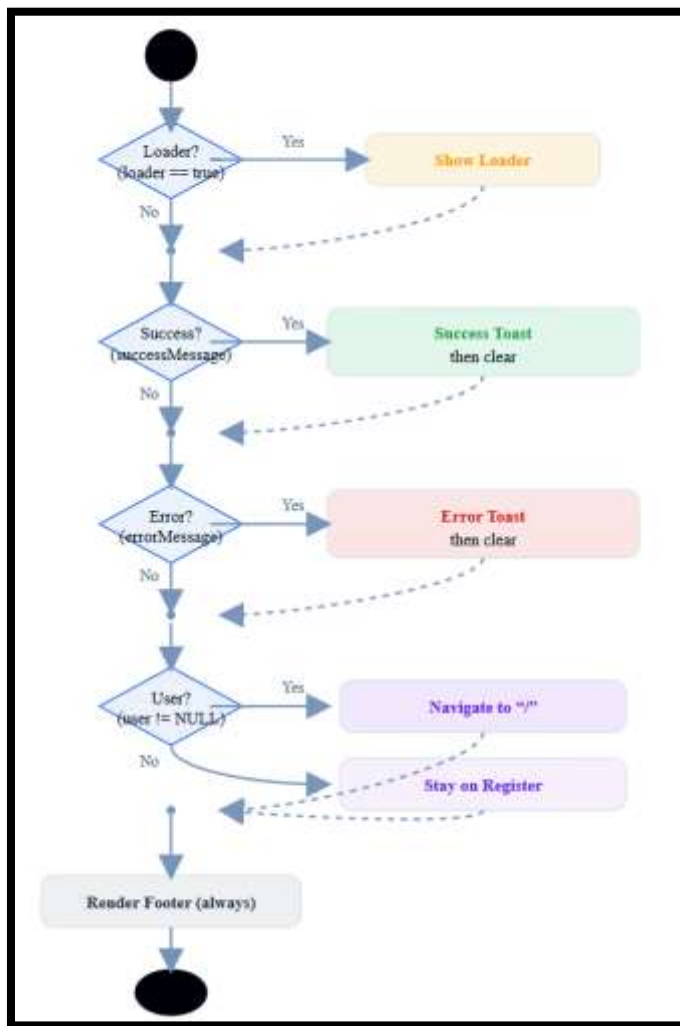


**Figure 10.3:** Cyclomatic Complexity for Login Module

### 5.1.2 Register Page

The Register Page applies similar logic to the Login Page, handling loaders, success/error messages, and redirecting upon successful registration.

- **Cyclomatic Complexity (CC): 5**
- **Independent Paths:**
  1. Loader → Success → Error → User → Navigate → Footer → End
  2. Loader → Success → Error → No User → Stay → Footer → End
  3. Loader → Success → No Error → User → Navigate → Footer → End
  4. Loader → Success → No Error → No User → Stay → Footer → End
  5. Loader → No Success → Error → User → Navigate → Footer → End



**Figure 11.1:** Decision-to-Decision Graph for Register Module

## Cyclomatic Complexity

**Decisions (d):** 4 → Loader? • Success? • Error? • User?

$$CC = d + 1 = 5$$

Also:  $CC = E - N + 2P = 5$  (single connected component,  $P = 1$ ).

## Independent Paths

1. P1: Loader → Success → Error → User → Navigate → Footer → End
2. P2: Loader → Success → Error → No User → Stay → Footer → End
3. P3: Loader → Success → No Error → User → Navigate → Footer → End
4. P4: Loader → Success → No Error → No User → Stay → Footer → End
5. P5: Loader → No Success → Error → User → Navigate → Footer → End

## Minimal Pseudo

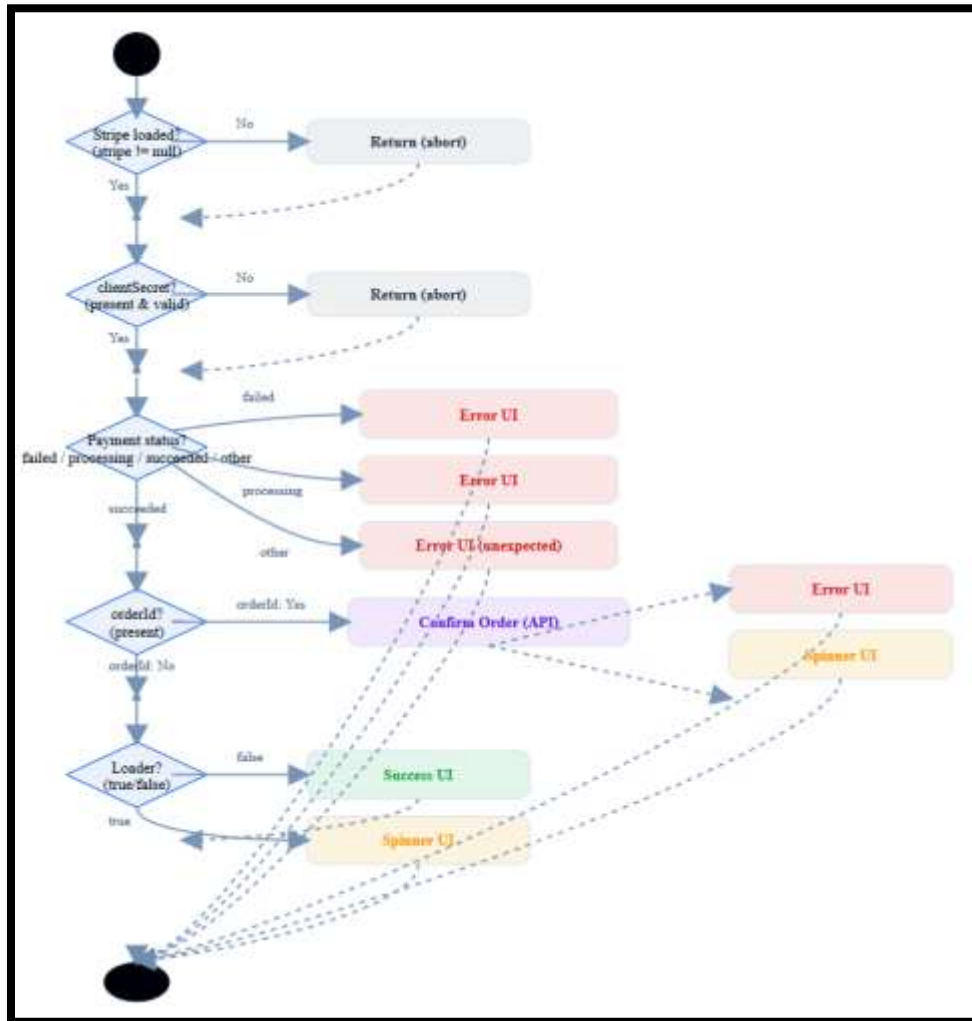
```
if (loader) { showLoader(); }
if (successMessage) {
  toastSuccess(successMessage); clearSuccess(); }
if (errorMessage) { toastError(errorMessage);
  clearError(); }
if (user) { navigate("/"); } else { /* stay */ }
renderFooter(); // always
```

**Figure 11.2:** Cyclomatic Complexity for Register Module

### 5.1.3 Confirm Order Page

The Confirm Order Page integrates with **Stripe Payment Gateway** and handles multiple conditions such as missing client secrets, payment statuses, and order confirmations.

- **Cyclomatic Complexity (CC):** 9
- **Independent Paths:**
  1. Stripe not loaded → Return → End
  2. Stripe loaded → No clientSecret → Return → End
  3. Stripe loaded → Valid clientSecret → Status Failed → Error UI → End
  4. Stripe loaded → Valid clientSecret → Status Processing → Error UI → End
  5. Stripe loaded → Status Succeeded + OrderId → Confirm → Error UI → End
  6. Stripe loaded → Status Succeeded + OrderId → Confirm → Spinner UI → End
  7. Stripe loaded → Status Succeeded → Loader False → Success UI → End
  8. Stripe loaded → Status Succeeded → Loader True → Spinner UI → End
  9. Stripe loaded → Status Unexpected → Error UI → End



**Figure 12:** Decision-to-Decision Graph for Confirm Order Module

## 5.2 Black Box Testing (User-Facing Functional Tests)

Black box testing focused on two critical **user-facing features**: **Customer Registration** and **Payment Confirmation**. The objective was to ensure that the system met functional requirements without inspecting internal code.

### 5.2.1 Customer Registration (Register Page)

**Preconditions:** Backend and Redux slices must be active.

**Test Scenarios:** Validation of name length, email format, and password rules.

**Table 2.1:** Black Box Test Cases – Customer Registration

Test ID	Input Condition	Expected Output	Pass/Fail Criteria
REG-01	Name = ""; Email valid; Password valid	Form blocked (HTML required)	Cannot submit form
REG-02	Name length = 1	Submit works	Form submits
REG-03	Name length = 50	Submit works	Form submits
REG-04	Name length = 51+	Error/blocked by backend	Error toast
REG-05	Email = ""	Blocked by browser validation	Cannot submit
REG-06	Email = "abc" (invalid)	Blocked	Cannot submit
REG-07	Email = "a@b.co"	Accepted	Submit works
REG-08	Email length = 254 chars	Accepted	Submit works
REG-09	Email length = 255+	Error	Error toast
REG-10	Password = ""	Blocked	Cannot submit
REG-11	Password length = 7	Blocked	Cannot submit
REG-12	Password length = 8	Accepted	Submit works
REG-13	Password length = 64	Accepted	Submit works
REG-14	Password length = 65+	Error	Error toast
REG-15	Valid inputs + backend success	Success toast + redirect to /	Meets criteria

REG-16	Valid inputs + backend failure	Error toast, stay on page	Handled
REG-17	Loader active during request	Spinner visible only async	Correct UI
REG-18	Double submit attempt	Second blocked	Only the first request processed

### 5.2.2 Payment Confirmation (Confirm Order Page)

**Preconditions:** Stripe and backend reachable.

**Test Scenarios:** Validation of Stripe initialization, clientSecret availability, and payment statuses.

**Table 2.2:** Black Box Test Cases – Payment Confirmation

Test ID	Input Condition	Expected Output	Pass/Fail Criteria
PAY-01	Stripe not loaded	Stay on spinner	Cannot proceed
PAY-02	ClientSecret missing	Stay on spinner	No crash
PAY-03	Succeeded + orderId present + backend 200 OK	Confirm → Success UI → orderId cleared	Correct UI
PAY-04	Succeeded + orderId missing	Success UI OR safe fallback	No crash
PAY-05	Payment status = processing	Error UI + back link	Correct UI
PAY-06	Payment status = requires_payment_method	Error UI + back link	Correct UI
PAY-07	Payment status = unexpected value	Default to error UI	Fallback works
PAY-08	Backend confirm API error	Loader doesn't persist forever → error shown	No infinite spinner

PAY-09	Network slow	Spinner persists until response	No UI freeze
PAY-10	OrderId present but backend confirm never returns	Spinner with timeout/error	Proper error path

## 5.3 Bug Detection and Solution

During testing, multiple bugs were identified and resolved. Each bug fix was followed by retesting to confirm successful resolution.

### 5.3.1 Customer Registration (Register Page)

- **REG-04 – Name length exceeds 50**
  - *Bug:* Names > 50 characters accepted → server errors.
  - *Fix:* Restricted length to 50 in both frontend and backend validation.
  - *Retest:* Input blocked; correct error shown.
- **REG-06 – Invalid email format**
  - *Bug:* Certain invalid emails bypassed validation.
  - *Fix:* Changed input type to “email” and applied stricter regex.
  - *Retest:* Invalid emails blocked.
- **REG-11 – Password length minimum**
  - *Bug:* 7-character passwords accepted.
  - *Fix:* Corrected rule to require  $\geq 8$  characters.
  - *Retest:* Now blocked correctly.
- **REG-18 – Double submit attempt**
  - *Bug:* Multiple submissions allowed.
  - *Fix:* Disabled button during first request; backend idempotency added.
  - *Retest:* Only first request processed.



#### 4. Bug Detection & Solution

Failed test cases, root causes, code fixes (before/after), and retest results.

**REG-04** Customer Registration **VALID Before**

**Symptom**  
Names with 51+ chars were accepted instead of blocked, causing server side error.

**Root Cause**  
• No max.length on input, frontend validator checked <= 50 instead of <= 50.

```
// BEFORE (React)
[username]
// validation
if (name.length < 1 || name.length > 50) errors.name = "Invalid";
```

**EXPECTED RESULT** Expected: "Error toast" for 51+ chars.

**Fix Summary**  
Add max.length=50    Align schema > 50    Test on deletion

```
// AFTER
[username]
// validation
if (name.length < 1 || name.length > 50) errors.name = "Name must be 1-50 chars";
```

**REG-06** Customer Registration **VALID Before**

**Symptom**  
Invalid email like "abc" could bypass client check with autoFocus.

**Root Cause**  
• Used type="text" and a weak regex.

```
// BEFORE
[email]
// JS
if (!/^[a-zA-Z0-9@]+$/i.test(email)) errors.email = "Invalid";
```

**EXPECTED RESULT** Expected: "Cannot submit".

**Fix Summary**  
Use type="email"    Stronger pattern    No autoFocus

```
// AFTER
[email]
// JS (optional extra)
const emailRe = /^[a-zA-Z0-9@]+$/i.test(email.trim());
if (!emailRe) errors.email = "Enter a valid email";
```

Figure 13.1: Customer Registration module bug and fix

**REG-11** Customer Registration **VALID Before**

**Symptom**  
Password length 7 was accepted due to off-by-one check.

**Root Cause**  
• Used > 7 instead of >= 8.

```
// BEFORE
if (password.length > 7) ok = true;
```

**EXPECTED RESULT** Expected: "Cannot submit".

**Fix Summary**  
Correct comparison    Unit test min length

```
// AFTER
if (password.length >= 8) ok = true;
```

**REG-18** Customer Registration **VALID Before**

**Symptom**  
Double submit sent two requests.

**Root Cause**  
• No submit lock / pending flag.

```
// BEFORE
Register
```

**EXPECTED RESULT** Expected: "Only first request processed".

**Fix Summary**  
Disable button during submit    (dependent backend option)

```
// AFTER (React)
handleSubmit ? "Submitting..." : "Register"
```

Figure 13.2: Customer Registration module bug and fix

### 5.3.2 Payment Confirmation (Confirm Order Page)

- **PAY-01 – Stripe not loaded**
  - *Bug:* Flow continued without Stripe initialized.
  - *Fix:* Added early return; spinner remains visible.
  - *Retest:* Process cannot proceed without Stripe.
- **PAY-02 – ClientSecret missing**
  - *Bug:* API called with undefined clientSecret.
  - *Fix:* Validation added before use.
  - *Retest:* Stable UI, no crash.
- **PAY-05 – Payment status = processing**
  - *Bug:* Incorrect spinner shown.
  - *Fix:* Explicit error UI for processing state.
  - *Retest:* Correct error UI displayed.
- **PAY-07 – Unexpected payment status**
  - *Bug:* Crash for unrecognized status.
  - *Fix:* Added default error handling.
  - *Retest:* Safe fallback implemented.
- **PAY-08 – Backend confirm API error**
  - *Bug:* Infinite spinner when API failed.
  - *Fix:* Error handling clears loader after failure.
  - *Retest:* Error shown, no infinite loop.
- **PAY-10 – OrderId confirm never returns**
  - *Bug:* No timeout caused permanent spinner.
  - *Fix:* Timeout mechanism introduced.
  - *Retest:* Proper error message shown.

**PAY-01** Payment Confirmation **STATUS: Defect**

**Symptom**  
Flow continued even when Stripe object was null.

**Root Cause**  
• Guard ran after side-effects.

```
// ✖ BEFORE
showSpinner();
const result = await handleStatus(stripe, clientSecret);
if (!stripe) return;
```

**EXPECTED BEHAVIOR** Expected: "Cannot proceed".

**Fix Summary**  
Daily release - Show spinner only while loading.

```
// ✔ AFTER
if (!stripe) { showSpinner(); return; }
const result = await handleStatus(stripe, clientSecret);
```

---

**PAY-02** Payment Confirmation **STATUS: Defect**

**Symptom**  
Missing clientSecret triggered fetch with empty load.

**Root Cause**  
• No null check before calling Stripe API.

```
// ✖ BEFORE
await stripe.retrievePaymentIntent(clientSecret);
```

**EXPECTED BEHAVIOR** Expected: "No crash".

**Fix Summary**  
Guard + return - Load and render on spinner.

```
// ✔ AFTER
if (!clientSecret) { showSpinner(); toastError("Missing client secret"); return; }
await stripe.retrievePaymentIntent(clientSecret);
```

Figure 13.1: Summary of Payment Bugs, Fixes, and Re-test Results

**PAY-03** Payment Confirmation **STATUS: Defect**

**Symptom**  
Spinner persisted forever on backend error.

**Root Cause**  
• No catch + finally to clear loader.

```
// ✖ BEFORE
setLoader(true);
const ok = await confirmOrder(orderId);
if (!ok) success();
```

**EXPECTED BEHAVIOR** Expected: "No infinite spinner".

**Fix Summary**  
try/catch/finally - Controlled loader control.

```
// ✔ AFTER
setLoader(true);
try {
  const ok = await confirmOrder(orderId);
  if (!ok) return error();
  return success();
} catch (e) {
  return error();
} finally {
  setLoader(false);
}
```

---

**PAY-10** Payment Confirmation **STATUS: Defect**

**Symptom**  
No timeout when confirm order returns.

**Root Cause**  
• Await without race/timeout; no abort controller.

```
// ✖ BEFORE
await confirmOrder(orderId); // could hang
```

**EXPECTED BEHAVIOR** Expected: "Proper error path".

**Fix Summary**  
Promise race timeout - AbortController yielding.

```
// ✔ AFTER
const timeout = {ok} => new Promise((_, reject) => setTimeout(() => reject(new Error("Timeout")), 1000));
await Promise.race([confirmOrder(orderId), timeout(1000)]);
```

Figure 13.2: Summary of Payment Bugs, Fixes, and Re-test Results

## Chapter 6: Deployment

### 6.1 Deployment Platform

The deployment of the *Multi-Vendor E-Commerce Marketplace* was carried out using **modern cloud-based hosting services** to ensure scalability, reliability, and ease of maintenance. The architecture was divided into **frontend** and **backend** deployments, hosted on different platforms optimized for their roles:

- **Frontend:** Deployed on **Vercel**, a platform designed for seamless hosting of Next.js applications. Vercel provides fast build times, global CDN distribution, and automated CI/CD pipelines. This ensures that users experience minimal latency regardless of their geographic location.
- **Backend:** Deployed on **Render**, which offers containerized application hosting with automatic scaling, HTTPS support, and database integration. Render's robust environment was used to host the **Node.js + Payload CMS backend**, ensuring secure API management and high uptime.
- **Domain & Hosting:** The main production website is hosted under a **custom domain** linked to Vercel for the frontend, with backend APIs securely connected via Render.

### 6.2 Deployment Process

The deployment process followed a **Continuous Integration and Continuous Deployment (CI/CD)** approach to streamline releases and reduce downtime. The process consisted of the following steps:

1. **Version Control:**
  - The source code was managed through **GitHub** repositories.
  - Developers pushed changes to feature branches, which were reviewed and merged into the main branch.
2. **Frontend Deployment (Vercel):**
  - Connected the GitHub repository to Vercel.
  - Each push to the `main` branch triggered an **automatic build and deployment**.
  - Vercel handled environment variables, build optimization, and global distribution via CDN.
3. **Backend Deployment (Render):**
  - The backend repository was linked with Render.
  - Render automatically built and deployed the backend service in a **Dockerized container**.
  - Environment variables for MongoDB, Stripe API keys, and authentication secrets were securely stored in Render's environment settings.
  - Auto-scaling and monitoring ensured backend stability under varying loads.
4. **Database Configuration:**

- The system uses **MongoDB Atlas**, a cloud-hosted NoSQL database.
  - The database was connected to the backend API with encrypted connections (TLS).
5. **Testing & Verification:**
- Post-deployment, both frontend and backend were tested for functionality, API connectivity, and performance.
  - Manual and automated regression tests were conducted before marking deployment successful.

## 6.3 Website URL

The production version of the system is accessible via the following link:

- **Main Website:** <https://bazarior.cloudmindsoftware.com/>
- **Backend (API Endpoint):** <https://bazarior-com-backend.onrender.com>
- **Frontend (Vercel App URL):** <https://bazariocom.vercel.app/>

## Chapter 7: Conclusion

### 7.1 Learnings

Throughout the development of the *Multi-Vendor E-Commerce Marketplace*, the team gained significant technical and managerial experience. Some of the key learnings include:

- **Full-Stack Development Skills:** Enhanced knowledge in building scalable applications using **Next.js, React, Node.js, Payload CMS, and MongoDB**.
- **Cloud Deployment Practices:** Learned to deploy and manage applications across **Vercel (frontend)** and **Render (backend)**, integrating them with a live custom domain and database services.
- **Secure Transactions:** Gained experience in implementing **Stripe Connect** for secure and efficient payment handling in a multi-vendor environment.
- **Project Collaboration:** Improved collaboration skills through **Git & GitHub**, enabling version control, teamwork, and code reviews.
- **Requirement Analysis & Testing:** Developed skills in applying **SRS methodologies**, creating use cases, and performing both **white box and black box testing** to ensure reliability and quality.

### 7.2 Limitations

Despite successfully implementing most planned features, certain functionalities could not be developed within the project timeline. The key limitations are as follows:

- **Communication Features:** A real-time **messaging system** between customers and vendors was not implemented. This limited direct interaction, which could enhance user engagement and trust.
- **Blog/Content Module:** A **blogging option** for vendors and admins, intended for publishing updates, promotions, and guides, was not completed. This restricted the platform's ability to foster a content-driven community.

### 7.3 Future Plan

For future development, the project can be extended with the following improvements:

- **Integrating Communication Tools:** Adding **in-app messaging or chat support** to allow real-time interaction between vendors and customers. This will improve customer service and strengthen vendor-customer relationships.

- **Content & Blog System:** Implementing a **blogging module** for vendors and administrators to publish articles, promotional campaigns, and product guides to boost engagement and SEO visibility.
- **Mobile Application:** Developing a **cross-platform mobile app** to extend usability and convenience for customers and vendors.
- **Advanced Analytics:** Incorporating **machine learning-based sales predictions, personalized recommendations, and vendor analytics** for smarter decision-making.
- **Enhanced Security Features:** Introducing **multi-factor authentication (MFA)** and advanced fraud detection to strengthen platform trust.
- **Scalability Upgrades:** Expanding infrastructure with a **microservices architecture** to better handle high traffic and vendor growth.

## Acknowledgment

We are truly grateful to everyone who supported us during the development of this project.

First, we would like to thank our respected instructor, **Yasin Sazid, Lecturer, Department of Computer Science and Engineering, East West University**, for his valuable guidance and continuous encouragement.

We also acknowledge the cooperation and teamwork of our group members, whose dedication and effort made this project possible. Working together has been a valuable learning experience for all of us.

*Thank You...*