TRƯỜNG ĐẠI HỌC SÀI GÒN

Khoa Công nghệ thông tin

Software Testing

Shopify System (Mow Garden)

Student: Nguyen Nhat Hai- 3122411046

Nguyen Thanh Dat- 3122411039

Nguyen Thanh Viet- 3122411245

Nguyen Thanh Trung Hieu- 3122411056

Tutor: PhD.Do Nhu Tai

Ho Chi Minh City, 19th December, 2025

Table of Contents

CHAPTER 1: OVERVIEW 5

1.1 Project Overview 7

1.2 Business Requirements 8

1.2.1 Non-Functional Requirements 8

1.2.2 Functional Requirements 9

CHAPTER 2: SYSTEM ANALYSIS AND DESIGN 11

2.1. System Design 11

2.1.1. Functionalities 11

2.1.2 User interface 28

2.1.3 Data design 29

2.2 Architecture Design 35

Decomposition View: C4 35

C1 – SystemContext 35

C2-Container 37

C3-Component(High-Level) 39

C3-Component(Model Level) 40

C4- Code/Implementation Diagram: 41

Deployment view 43

Block Diagram 44

CommunicationView 44

CHAPTER 3: Test Plan 46

3.1. Introduction 46

3.1.1. Purpose 46

3.1.2. Definitions, Acronyms, and Abbreviations 46

3.1.3. References 46

3.1.4. Background information 46

3.1.5. Scope of testing 47

3.1.6. Constraints 47

3.1.7. Risk List 47

3.1.8. Training Needs 48

3.2. Requirements for Testing 48

3.2.1. Test Items 48

3.2.2. Acceptance Test Criteria 49
3.3.1. Functionality 49
3.3.2. Usability 52
3.3.3. Design Constraints 53
3.3.4. Interfaces 54
3.4. Features Not to Be Tested 54
3.4.1. Functional Features 55
3.4.2. Non-Functional Features 55
3.5.2. Test Stages 60
3.5.3. Test Phases 61
3.6. Resource 62
3.6.1. Human Resource 62
3.6.2. Test Management 62
3.7. Test Environment 63
3.7.1. Hardware 63
3.7.2. Software 63
3.7.3. Infrastructure 64
3.8. Test Milestones 64
3.9. Deliverables 65
CHAPTER 4: TEST DESIGN 67
4.1. Objectives of Test Design 67
4.2. Basis for Test Design 67
4.3. Testing Methodologies and Techniques Applied to the Checkout Module 67
4.3.1. Black-box Testing Methodology 67
4.3.1.2. Black-box Testing Techniques and Test Case Generation 68
4.3.2. White-box Testing Methodology 76
4.3.3. Workflow-based Business Testing (End-to-End Testing) 79
a) Complete Order Workflow (User Journey) 81
b) Form Validation and User Experience Workflow 82
c) User Interface State Handling Workflow 82
d) System Navigation Workflow 82
e) Error Notification and System Feedback Workflow 83
f) User Interface Consistency Workflow 83

4.3.3.5. Tools and Implementation 83
4.3.3.6. Test Results 83
4.3.3.7. Conclusion 84
4.3.4. Automated Testing and Unit Testing 84
4.3.4.1. Concept of Automated Testing 84
4.3.4.2 Unit Testing in the MOW Garden Project 85
1. Objectives of Unit Testing 85
2. Scope of Unit Testing 85
3. Detailed Unit Testing Content 86
3.1. User Module 86
3.2. Product Module 86
3.3. Cart Module 86
3.4. Order Module 87
3.5. Validation and Error Handling 87
4. Characteristics of Unit Testing in the Project 87
5. Conclusion 87
4.4. Application of GenAI in Test Design 88
4.4.1. Objectives of Applying GenAI in Test Design 88
4.4.2. Technologies and Tools Used 89
4.4.3. GenAI-based Test Scenario Generation Process 89
Step 1: Select the Target Controller 89
Step 2: Execute the Test Scenario Generation Command 89
Step 3: Generate Test Scenarios 90
Step 4: Storing Generated Test Scenarios 90
4.4.4. Role of Test Scenario Files in Test Design 91
4.4.5. Scope of Application and Limitations 92
Limitations 92
4.4.6. Conclusion 92
CHAPTER: Testing Result 93
5.1. Illustration of Test Execution 93
5.1.1. Test Execution Scope 93
5.2.2. Tools and Test Execution Environment 93
5.2.3. Illustration of Test Execution 94

5.3. Test Report 103
5.4. Defect Report 105
5.5. Test Summary Report 106
5.5.1. Requirement Fulfillment Evaluation 106
5.5.2. Test Design Effectiveness Evaluation 107
5.5.3. Limitations and Future Improvements 107
5.6. Conclusion 107

Index image

Pic 1. Bussiness context diagram 13
Pic 2 Checkout 14
Pic 3 Order 15
Pic 4 Checkout process 16
Pic 5 Order Process 16
Pic 6 . Usecase view diagram 17
Pic 7. Actors take in systems 17
Pic 8. User control function decomposition diagram 19
Pic 9 . Product catalog functional decomposition diagram 19
Pic 10. Cart functional decomposition diagram 20
Pic 11. Checkout functional decomposition diagram 20
Pic 12. Order functional decomposition diagram 21
Pic 13. Inventory functional decomposition diagram 21
Pic 14 Checkout screen 28
Pic 15 Order screen 29
Pic 16. Conceptual Model 29
Pic 17. Conceptual ERD 33
Pic 18. Logical ERD 33
Pic 19. C1 – SystemContext 36
Pic 20. C2-Container 37
Pic 21. C3-Component(High-Level) 39
Pic 22. C3-Component(Model Level) 40
Pic 23. C4- Code/Implementation Diagram 41
Pic 24. Deployment view 43
Pic 25. Block Diagram 44
Pic 26. CommunicationView 45
Pic 27 Checkout / Place Order 78
Pic 28 e2e 81
Pic 29 pass e2e 84
Pic 30 .unit tess 88
Pic 31 genAI 90
Pic 32 test scenario 91

Pic 33 npm test 98
Pic 34 Scenario 1 Jmeter 100
Pic 35 jMeter Result 101
Pic 36 Scenario 2 jMeter 102
Pic 37 Scenario 2 jMeter Result 103

CHAPTER 1: OVERVIEW

1.1 Project Overview

The **MOW Garden Website** project is an online ornamental plant e-commerce system developed to connect buyers and administrators through a web-based platform. The system aims to create a modern, intuitive, and user-friendly shopping environment while effectively supporting product, order, and inventory management for administrators.

Users can easily search for products, view detailed product information, place orders, and track order statuses. Meanwhile, administrators have full control over system operations, including managing products, categories, orders, inventory, and user accounts.

The primary objective of the system is to provide a specialized e-commerce platform for the ornamental plant industry while automating the entire purchasing process—from product browsing, ordering, and payment to inventory updates and stock restoration in case of order cancellation. This automation helps minimize errors caused by manual operations, improve operational efficiency, and ensure data consistency throughout the system.

In addition, the system places strong emphasis on user experience by offering a friendly interface and stable performance across multiple devices, including desktop computers and mobile phones.

The system targets two main user groups. The **Buyer** group consists of individual customers who access the website to browse products, purchase ornamental plants, complete payments, and track their order status. The **Admin** group is responsible for managing the entire system, including product management, category management, order processing, inventory control, and user management. Each user group is clearly assigned specific permissions to ensure system security and stable operation.

Regarding implementation scope, the system is developed as a web application with a Single Page Application (SPA) interface using **ReactJS**. The backend is built with **Node.js** and **Express**, while **MongoDB** is used as the database for data

storage and management. With this architecture, the MOW Garden Website is highly scalable and can be extended in the future to integrate online payment gateways or delivery systems, thereby enhancing the system's practical value and applicability.

1.2 Business Requirements

1.2.1 Non-Functional Requirements

No.	Requirement Type	Detailed Description
1	Performance	The system responds to user requests within less than 3 seconds with up to 500 concurrent users.
2	Security	Passwords are securely hashed. Only authorized users can access the admin dashboard. All communications are conducted over HTTPS.
3	Reliability	The system maintains an uptime of at least 99% in the testing/staging environment.
4	Scalability	The architecture supports horizontal scaling to handle increased user load.
5	Usability	The interface is simple, clear, supports Vietnamese with diacritics, and is compatible with both desktop and mobile devices.

6	Maintainability	The source code is modularized (Access Control, Catalog, Order, Inventory, etc.) and accompanied by API documentation.
7	Compatibility	The system operates stably on Chrome, Edge, Safari, Firefox, and common operating systems.
8	Backup & Recovery	Data is backed up daily and can be restored in case of system failure.
9	Localization	Supports Vietnamese with diacritics, special characters, and VND currency symbols.

1.2.2 Functional Requirements

No.	Function	Detailed Description
1	User Management (Access Control)	- Users can register, log in, and log out. - Admin can lock/unlock user accounts. - Role-based access control for Buyer and Admin.
2	Product Catalog	- Buyers can browse product categories, filter by type/price, and view product details. - Admin can create, read, update, and delete products and allocate inventory.

3	Shopping Cart	- Buyers can add, remove, and update products in the cart. - The system automatically calculates total cost and shipping fees.
4	Checkout / Payment Process	- Buyers enter shipping information and select payment methods (COD). - The system confirms orders and updates inventory accordingly.
5	Order Management	- Buyers can view order lists and order details. - Admin can track and update order statuses.
6	Inventory Management	- Admin can add, update, and delete products. - Inventory is automatically deducted upon successful checkout.

CHAPTER 2: SYSTEM ANALYSIS AND DESIGN

2.1. System Design

2.1.1. Functionalities

2.1.1.1. Business Context

The MOW Garden website operates under an online retail model with the following core business modules: Product Catalog, Shopping Cart, Checkout Process (COD), Order Management, Inventory, and Access Control. These modules work together to form a complete business flow, from product browsing and ordering to inventory management and system administration.

Access Control

Customers are required to register an account or log in in order to perform shopping cart and ordering operations. After successful registration or login, users are redirected to the product catalog page.

Administrators use pre-created system accounts to log in to the admin dashboard, where they can manage products, inventory, orders, and lock or unlock user accounts. If a customer account is locked, the system will prevent the user from logging in and from creating any orders.

Product Catalog

Customers can browse the list of plants, filter products by category or price, and view detailed information for each product. On the product detail page, the system displays comprehensive information including plant name, description, images, selling price, and current inventory quantity.

Administrators (Admin) can add new products, update product information, change images, or delete products. When an Admin adds a new product, the system automatically creates a

corresponding inventory record with an initial quantity of 0. Each product has exactly one inventory record, and the system does not support multiple inventory locations.

Shopping Cart

Customers can add products to the cart from either the product listing page or the product detail page. The shopping cart displays the selected products along with quantity, images, unit price, shipping fee, and total order value, which is calculated as follows:

- Subtotal = total product price
- Shipping fee = fixed value defined by the system
- Total amount = subtotal + shipping fee

The shopping cart is stored per user in `user.cartData`. Any add or remove operation updates the cart data, and the user interface reflects the correct total amount in real time.

Checkout Process (COD)

When the customer selects "Checkout," the system displays the cart information and requests delivery details, including recipient name, delivery address, email, and phone number. After confirmation, the frontend sends the product list and final total amount (including shipping fee) to the backend.

The backend verifies inventory availability for each product. If sufficient stock exists, the system creates a new order with the COD payment method, stores the total amount received from the frontend, deducts inventory quantities, and clears the user's shopping cart. A successful order confirmation message is then returned and displayed on the user interface.

Order Management

Customers can view their order history, including product list, quantities, prices, total amount, delivery address, and order status. When viewing order details, the system displays the order ID, order date, total amount (including shipping fee), product list (snapshot at the time of purchase), and delivery status.

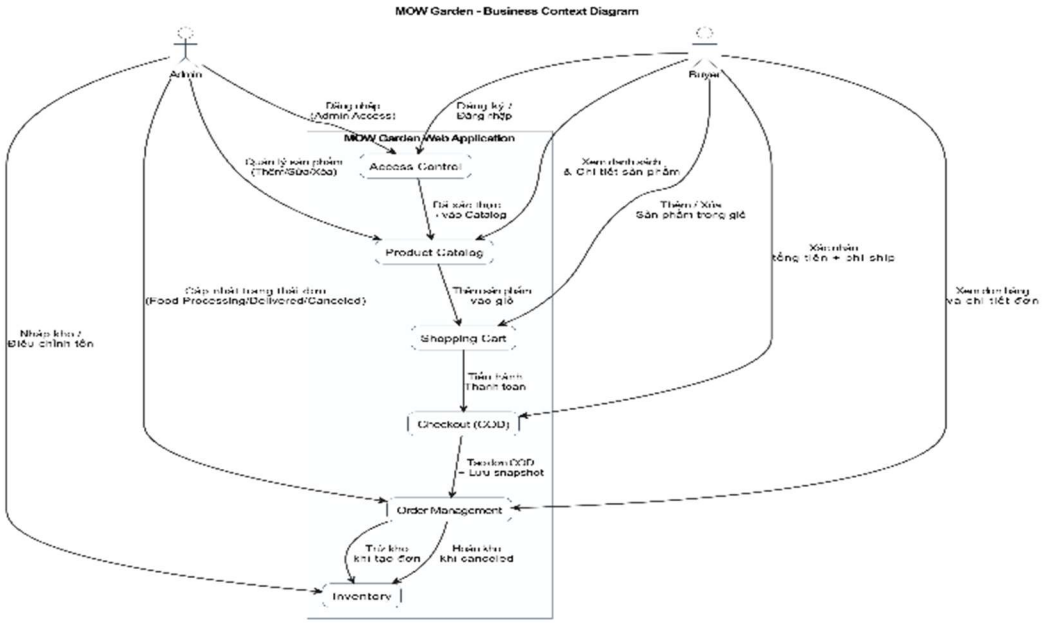
Administrators can view all customer orders and update order statuses. The system supports the following valid order states:

1. **Food Processing** – order has been created and is being processed
2. **Out for Delivery** – order is currently being delivered

3. **Delivered** – delivery completed and COD payment collected
4. **Canceled** – order is canceled and inventory is restored

Inventory Management

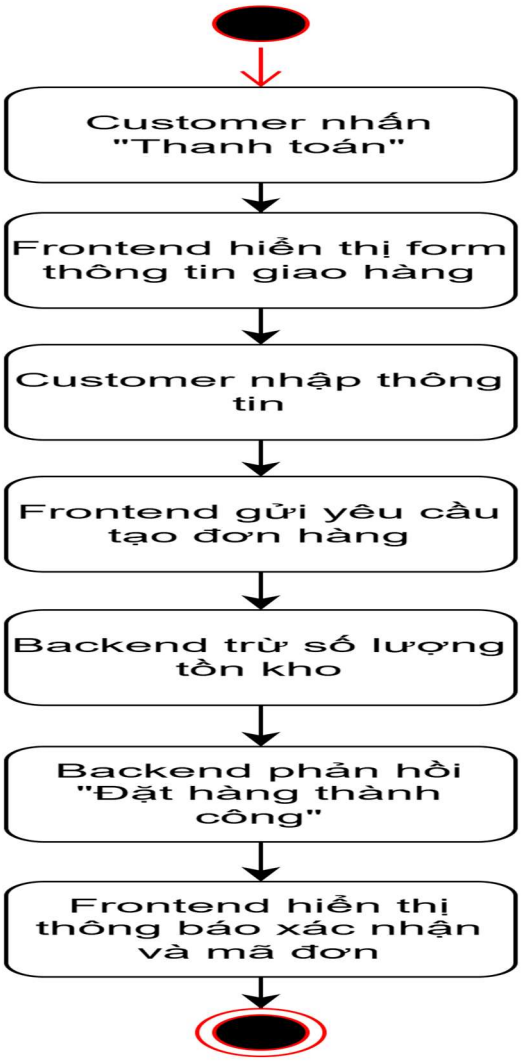
Each product has a single inventory record. When an order is successfully created, inventory quantities are deducted accordingly; when an order is canceled, inventory is restored. Administrators can manually adjust inventory quantities or delete inventory records when removing products from the catalog. Inventory data is maintained consistently and synchronized with all purchase and cancellation operations.



Pic 1. Bussiness context diagram

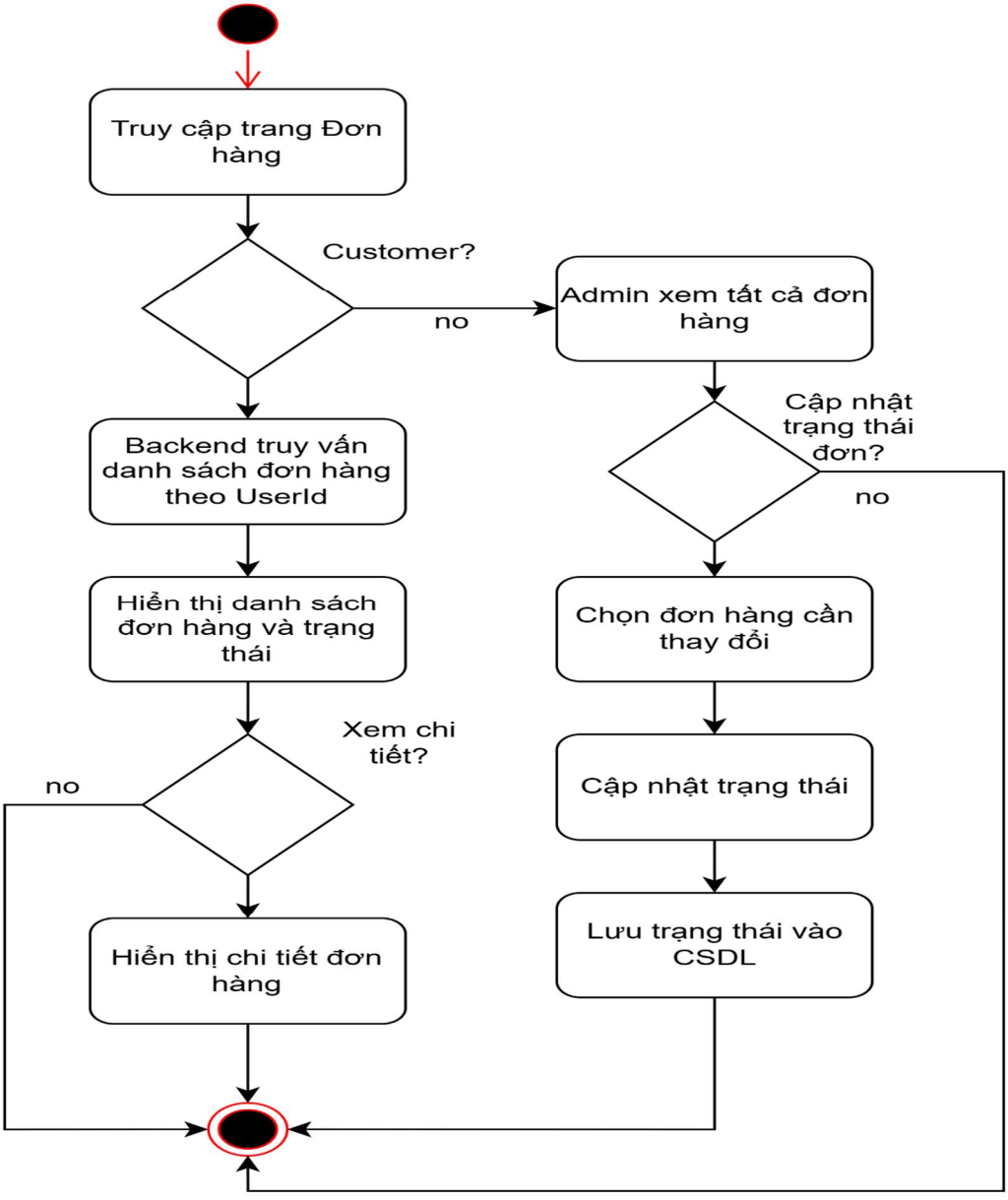
2.1.1.2 Work flow

Checkout



Pic 2 Checkout

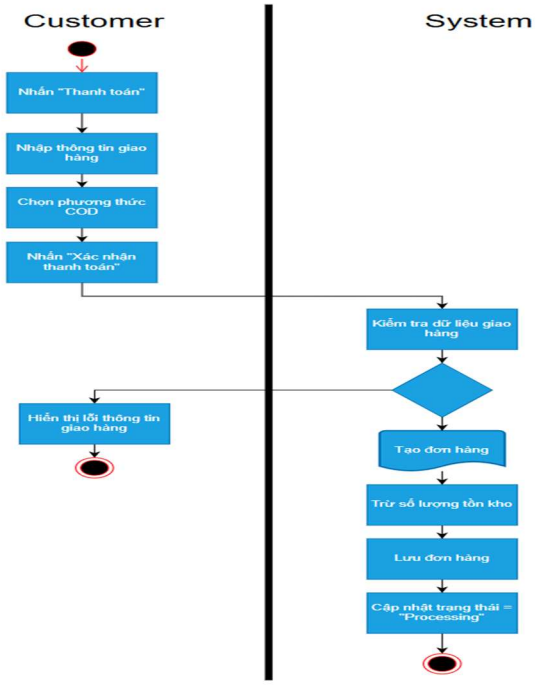
Order



Pic 3 Order

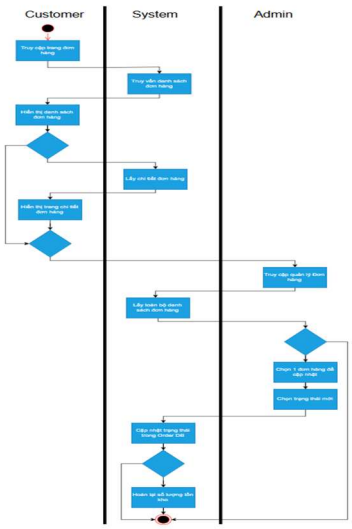
2.1.1.3 Sơ đồ quy trình nghiệp vụ

Checkout Process



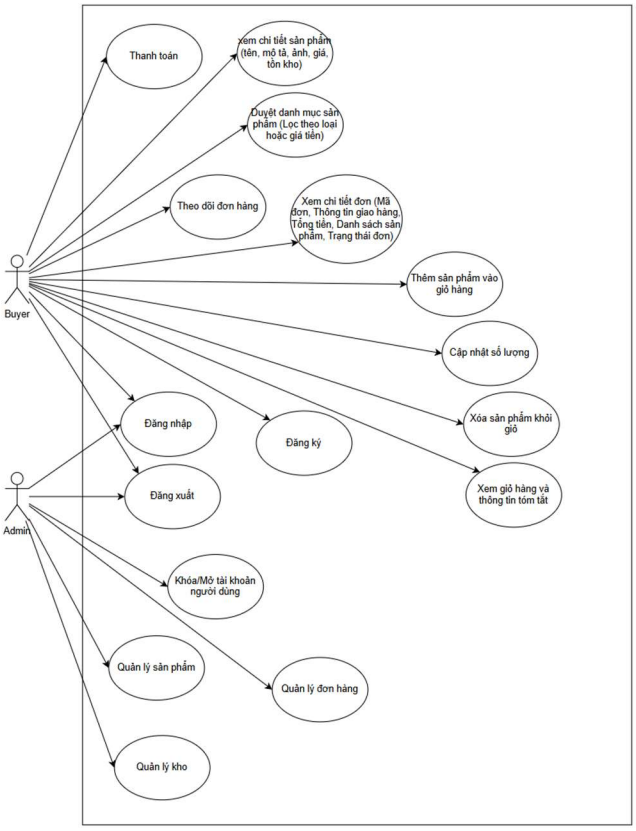
Pic 4 Checkout process

Order Process



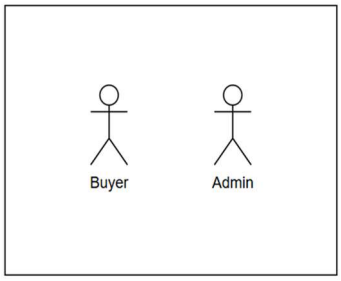
Pic 5 Order Process

2.1.1.4 Usecase view



Pic 6 . Usecase view diagram

Actors take in systems



Pic 7. Actors take in systems

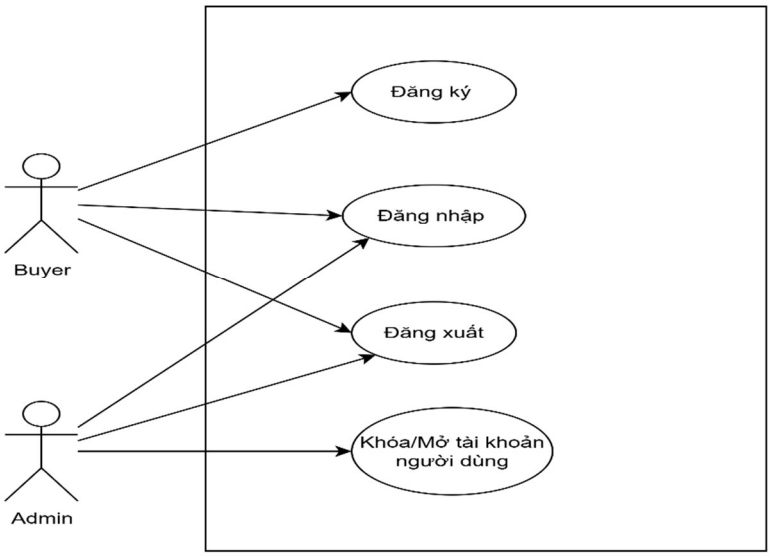
- **Buyer:** Search for products, add items to the cart, complete checkout, and track their own orders.
- **Admin:** Operate and manage the system, including products, inventory, orders, and users.
- **Payment Provider:** Process non-cash payment transactions.

Main Functionalities

- **Access Control:**
  This function allows users to self-register accounts (Buyer). Users (Buyer/Admin) can log in and are redirected to the appropriate area (Product Catalog or Admin Dashboard). Users can securely log out, and Admins can lock or unlock user accounts.
- **Product Catalog:**
  This function allows Buyers to browse product categories, filter products by type or price, and view detailed product information (name, description, images, price, and stock availability). Admins are authorized to Add / Update / Delete products and assign products to inventory.
- **Shopping Cart:**
  This function allows Buyers to add products to the cart from the Catalog or Product Detail page, view the cart summary, update product quantities, or remove products. The system automatically recalculates the total amount (unit price, shipping fee, and order total) and navigates the user to the Checkout process.
- **Payment Process:**
  This function allows Buyers to complete orders by entering and confirming order information, selecting the payment method (COD), and confirming payment. The system generates an invoice and associates it with the Order, deducts inventory quantities per item, restores inventory when an order is canceled, and sends order confirmation notifications.
- **Order:**
  This function allows Buyers to view their order list and access detailed order information (order ID, delivery information, total amount, product list, and order status). Admins can view all customer orders, monitor them, and update order statuses.
- **Inventory:**
  This function allows Admins to manage inventory quantities, including stock-in operations, updating quantities by category or by individual product, and removing products from inventory.

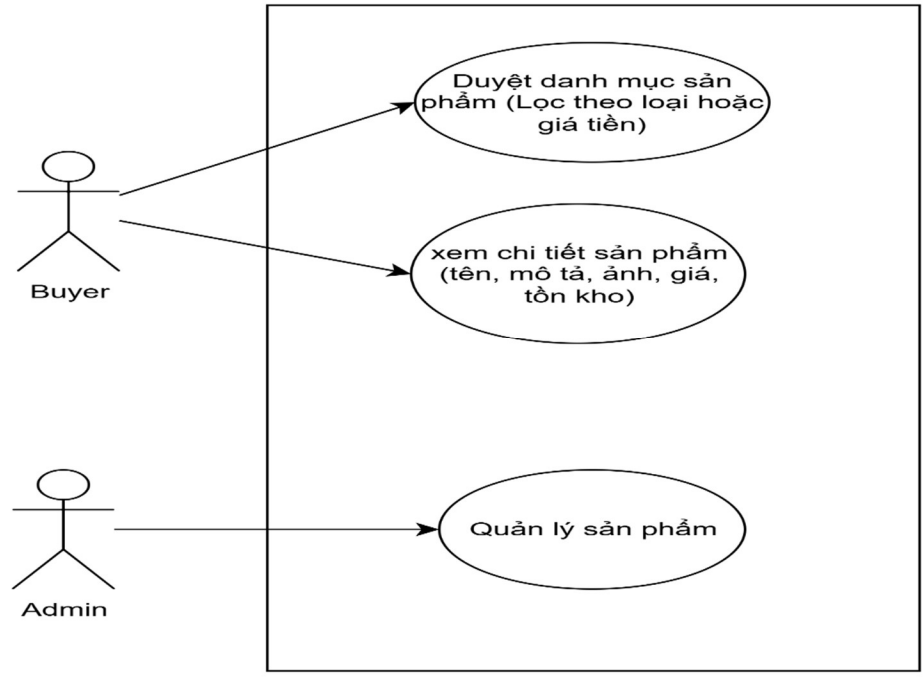
Functional Decomposition Diagram of the System

- **User Access Control**



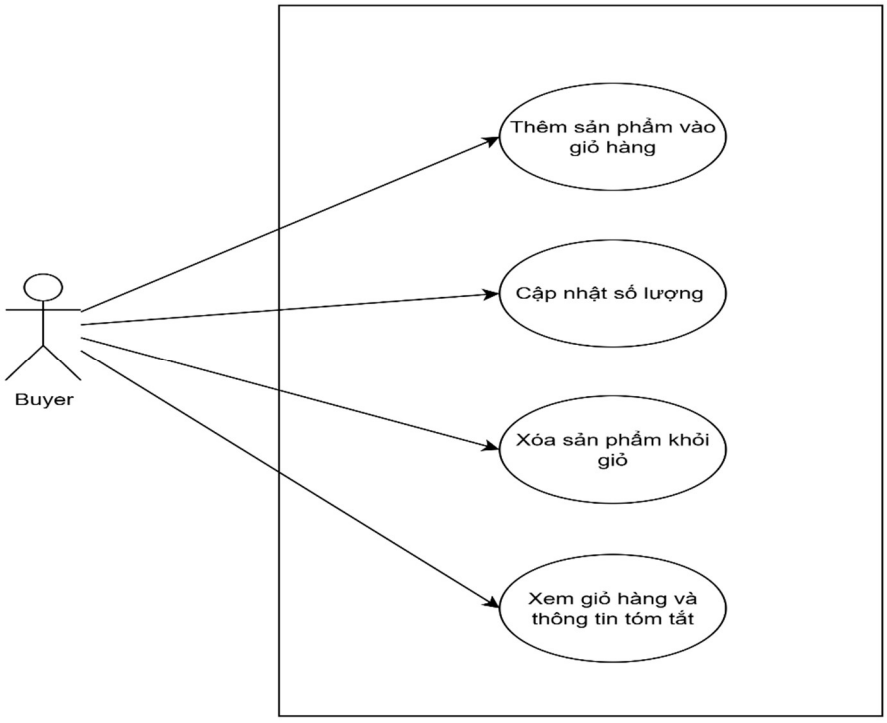
Pic 8. User control function decomposition diagram

- Danh mục sản phẩm



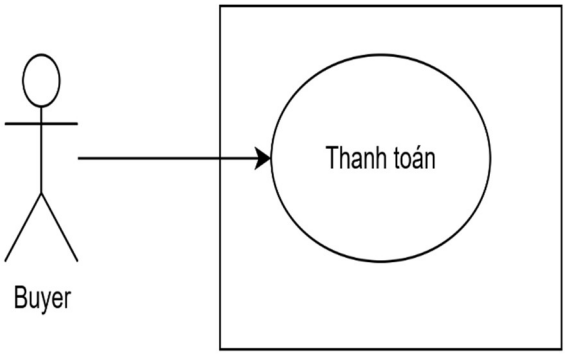
Pic 9 . Product catalog functional decomposition diagram

- Cart



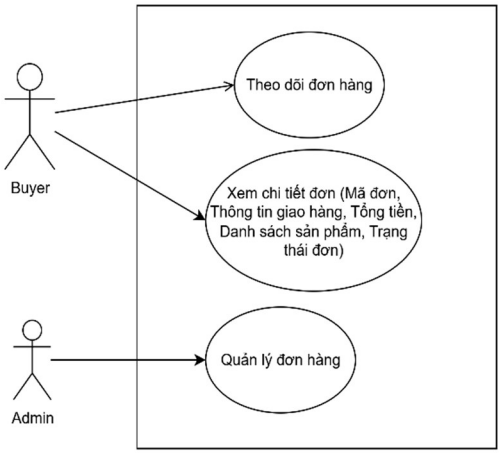
Pic 10. Cart functional decomposition diagram

- Checkout Process



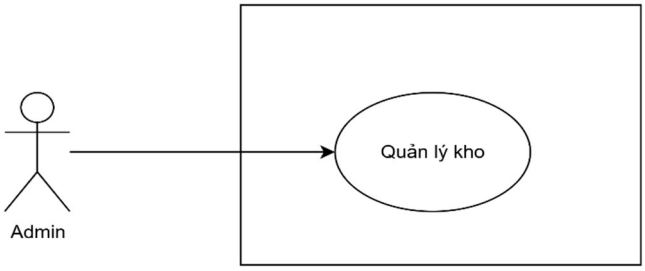
Pic 11. Checkout functional decomposition diagram

- Order



Pic 12. Order functional decomposition diagram

- Inventory

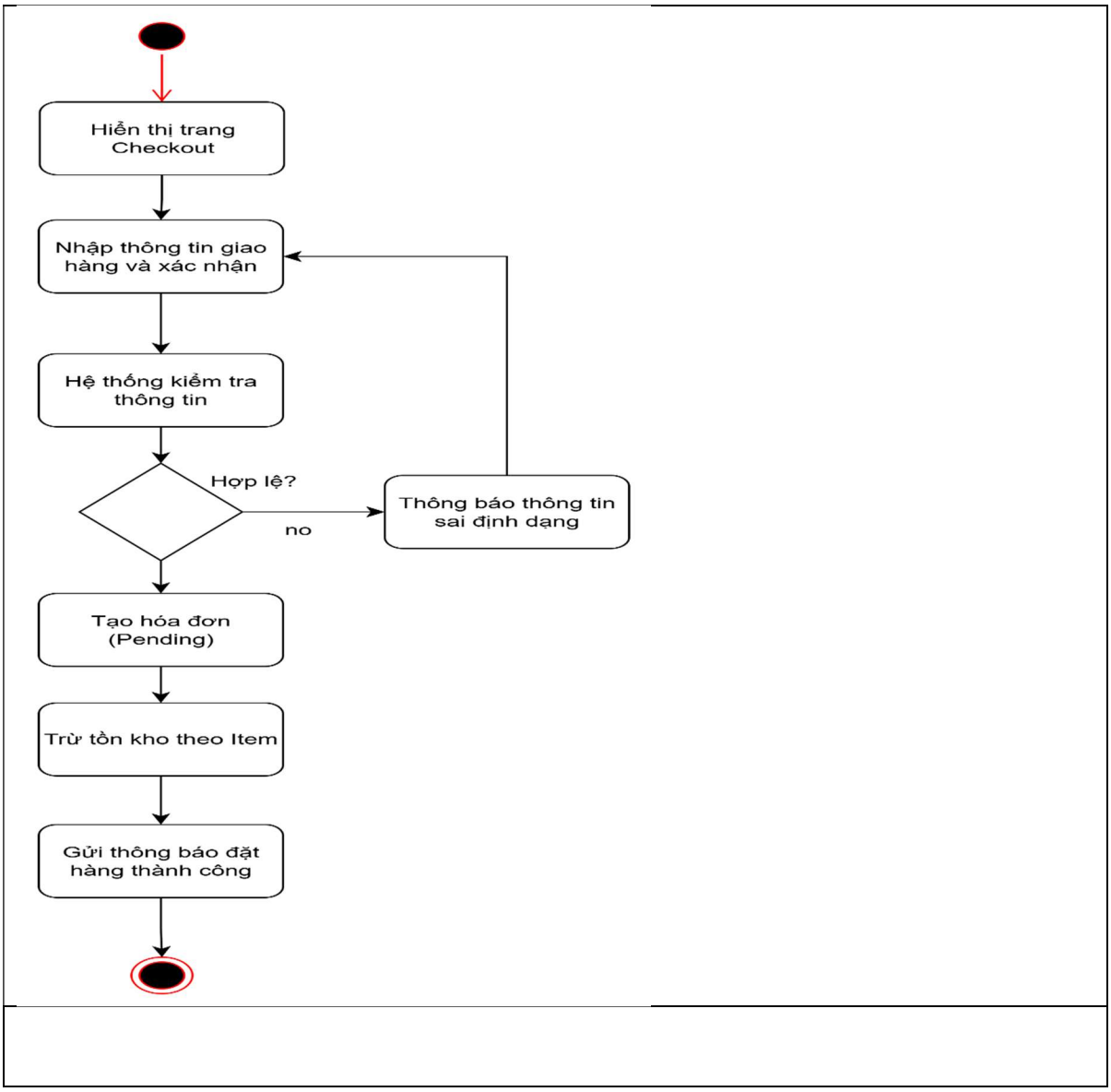


Pic 13. Inventory functional decomposition diagram

USECASE SPECIFICATIONS Checkout

UC 4		Checkout Process
Mô tả		Chức năng này cho phép Buyer hoàn tất đơn hàng, nhập và xác nhận đơn hàng, chọn phương thức thanh toán (COD), xác nhận thanh toán. Hệ thống tạo hóa đơn và gắn vào Order, trừ tồn kho theo Item và gửi thông báo xác nhận.
Tác nhân	Chính	Buyer
	Phụ	Hệ thống
Tiền điều kiện		- Buyer đi từ Checkout (giỏ đã hợp lệ). - Phương thức thanh toán được cấu pic hoạt động.

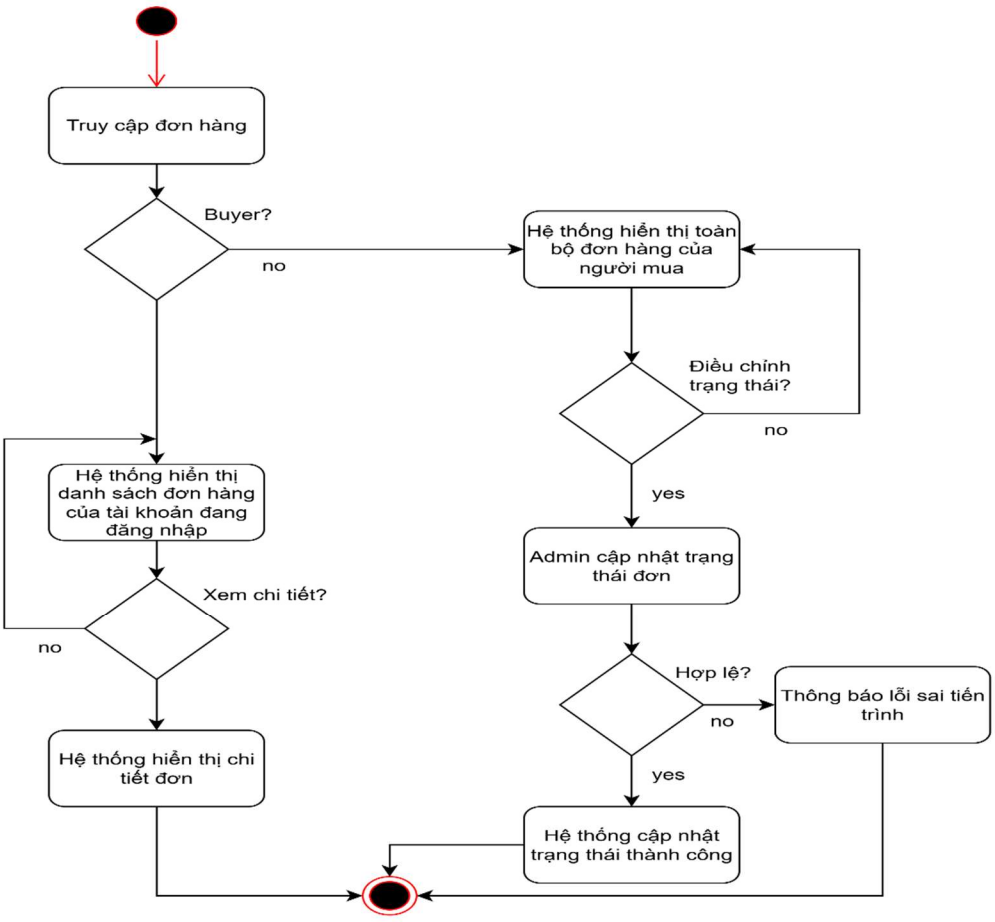
<table>
<tr><td colspan="2"></td><td>- Chính sách tồn kho đã xác định thời điểm trừ tồn sau khi Order được xác nhận.</td></tr>
<tr><td rowspan="2">Hậu điều kiện</td><td>Thành công</td><td>- Đơn hàng ở trạng thái phù hợp.
- Inventory được trừ theo từng Item trong Order và có lịch sử biến động.
- Buyer nhận thông báo xác nhận</td></tr>
<tr><td>Lỗi</td><td>- Thông tin giao hàng không hợp lệ thông báo lỗi yêu cầu nhập lại.
- Hết hàng hoặc giá bị thay đổi lúc chụp đơn hiển thị khác biệt và yêu cầu Buyer xác nhận lại.
- Thanh toán thất bại cho Buyer chọn lại phương thức.
- Trừ tồn thất bại thông báo lỗi.</td></tr>
<tr><td colspan="3">Đặc tả chức năng</td></tr>
<tr><td colspan="3">Luồng sự kiện chính</td></tr>
<tr><td colspan="3">Chức năng bắt đầu khi Buyer muốn Thanh toán:
- Kiểm tra và chụp đơn
+ Hệ thống chụp giỏ, tạo Order, tính lại giá/ship/tổng đơn.
+ Nếu phát hiện hết hàng hoặc giá thay đổi hiển thị cho Buyer xác nhận lại.
- Thông tin giao hàng
+ Buyer nhập và xác nhận tên, địa chỉ, email, SĐT.
+ Hệ thống xác nhận định dạng.
- Chọn phương thức
+ Buyer chọn COD.
- Xác nhận thanh toán.
+ COD
* Hệ thống tạo hóa đơn (Pending/COD).
* Trừ tồn kho theo từng Item, ghi nhận biến động.
* Gửi thông báo xác nhận cho Buyer.</td></tr>
<tr><td colspan="3">Sơ đồ hành động</td></tr>
</table>



USECASE SPECIFICATIONS Order

UC 5		Order
Mô tả		Chức năng này cho phép (Buyer) xem danh sách đơn của mình, tra cứu chi tiết đơn (mã đơn, thông tin giao hàng, tổng tiền, danh sách sản phẩm, trạng thái). (Admin) xem tất cả đơn của người mua, giám sát và cập nhật trạng thái.
Tác nhân	Chính	Buyer, Admin
	Phụ	Hệ thống

<table>
<tr><td colspan="2">Tiền điều kiện</td><td>- Người dùng đăng nhập (Buyer xem đơn của chính mình, Admin có quyền quản trị).
- Đơn hàng đã được tạo từ Checkout.</td></tr>
<tr><td rowspan="2">Hậu điều kiện</td><td>Thành công</td><td>- Buyer xem được danh sách, chi tiết chính xác.
- Admin cập nhật trạng thái đúng quy tắc, hệ thống ghi nhật ký kiểm tra và gửi thông báo.</td></tr>
<tr><td>Lỗi</td><td>- Buyer không thể truy cập đơn hàng thông báo lỗi.
- Chuyển trạng thái không hợp lệ sai tiến trình thông báo từ chối và hiển thị tiến trình hợp lệ.</td></tr>
<tr><td colspan="3">Đặc tả chức năng</td></tr>
<tr><td colspan="3">Luồng sự kiện chính</td></tr>
<tr><td colspan="3">- Chức năng này bắt đầu khi Buyer muốn xem lại đơn hàng hoặc Admin muốn xem đơn của người mua và theo dõi tiến trình đơn.
- Buyer xem danh sách hoặc chi tiết đơn hàng
+ Buyer mở Đơn hàng của tôi.
+ Hệ thống hiển thị danh sách đơn.
+ Buyer nhấn Xem chi tiết một đơn.
+ Hệ thống hiển thị chi tiết đơn gồm: Mã đơn, Thông tin giao hàng, Tổng tiền, Danh sách sản phẩm, trạng thái hiện tại.
- Admin quản lý và giám sát
+ Admin chọn Quản lý đơn hàng.
+ Hệ thống hiển thị toàn bộ đơn có thông tin cơ bản Trạng thái, Ngày, Khách.
+ Admin chọn đơn để xem chi tiết.
+ Admin chọn cập nhật trạng thái theo quy tắc: Processing, Out for delivery, Deveried.
+ Hệ thống cập nhật trạng thái và ghi nhận.</td></tr>
<tr><td colspan="3">Sơ đồ hành động</td></tr>
</table>



Giao diện minh họa

2.1.1.5 User Story

Access Control

- Each Buyer / SysAdmin is considered a system user (User).
- As a Buyer or SysAdmin, I want to log in to the system.
- When a Buyer logs in, they are redirected to the Product Catalog page.
- When a SysAdmin logs in, they are redirected to the System Administration page.
- As a Buyer or SysAdmin, I want to log out of the system.
- As a SysAdmin, I want to be able to lock / unlock user accounts according to system policies.

Product Catalog

- As a Buyer, I want to view the product list on the homepage and be able to filter products by category or price range, so that the displayed list is narrowed down to only relevant items.
- As a Buyer, I want to navigate to the product detail page to view basic information such as product name, description, price, and current stock quantity at each store/warehouse.
- As a System Administrator (SysAdmin), I want to manage products by performing CRUD operations (Create, Read, Update, Delete) and assign existing inventory to each product.

Shopping Cart

- As a Buyer, I want to add any product from the catalog page to the shopping cart (default quantity = 1).
- As a Buyer, I want to add products from the product detail page to the shopping cart (default quantity = 1).
- As a Buyer, I want to view the products added to the cart along with a summary table, including: cart subtotal, estimated shipping fee, and total order amount.
- As a Buyer, I want to update the quantity of each product in the cart.
- When the product quantity changes, the summary table must be updated accordingly.
- As a Buyer, I want to remove any product from the shopping cart.
- When a product is removed, the summary table must also be updated accordingly.
- As a Buyer, I want to proceed to checkout for my shopping cart.
- When the total quantity of products in the cart equals 0, the checkout process must not be allowed.
- When the cart is eligible for checkout, the checkout process is initiated.

Checkout Process

- Any Buyer is allowed to proceed with checkout.

- When the checkout process starts:
  - The system validates product information (price and inventory availability).
  - The system processes the payment (via COD).
  - The system sends notifications to the Buyer.
- If any product is invalid (out of stock or price has changed), the checkout process is canceled and the Buyer is notified to reconfirm the order.
- If the payment is successful, the system will:
  - Mark the shopping cart as paid
  - Generate an invoice and attach it to the corresponding order
  - Deduct inventory quantities for each product
  - Send payment confirmation to the Buyer
- If the payment fails or is canceled, the system will:
  - Mark the payment as failed
  - Allow the Buyer to select an alternative payment method

Order

- As a Buyer, I want to view my order list and be able to filter orders by status.
- As a Buyer, I want to view order details, including:
  - Order ID
  - Delivery information
  - Product list
  - Subtotal / shipping fee / total amount
  - Current status and status change history
- As a System Administrator (SysAdmin), I want to view all customer orders with advanced filters (by status and buyer).
- As a System Administrator (SysAdmin), I want to update order status following the valid workflow:

  Processing → Out for Delivery → Delivered → Canceled

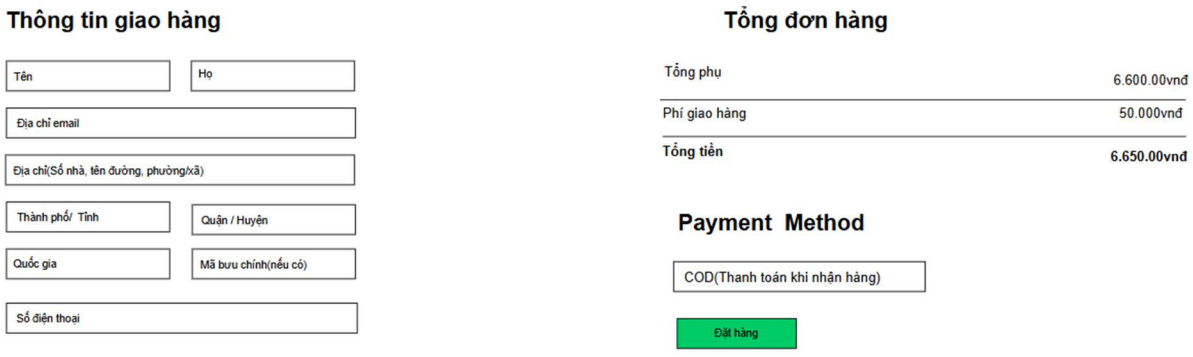
- When the order status changes, the system must:
  - Record an Audit Log (operation history)

Inventory

- As a System Administrator (SysAdmin), I want to add products (IN) to a selected warehouse from the catalog or product list, including quantity and reason for stock-in.
- As a System Administrator (SysAdmin), I want to adjust inventory quantities (±) for products by category or by individual items, along with a reason (stock-taking, damage, warehouse transfer).
- As a System Administrator (SysAdmin), I want to remove a product from the warehouse only when the stock quantity equals 0 and there are no reserved orders.
- When inventory levels change, the system must create an **Inventory Movement** record that includes:
    - Reference (ref)
    - Actor (performed by)
    - Accurate and atomic inventory quantity updates
- When an order is confirmed or paid according to policy, the system must generate an **OUT (stock-out)** movement for each product in the order.
- When an order is confirmed as canceled, the system must restore inventory quantities for each product in the order.

2.1.2 User interface

2.1.2.1 Checkout screen



Pic 14 Checkout screen

This screen appears after adding products to the shopping cart and the user clicks "Proceed to Checkout". The user then enters all the required information and uses the COD (Cash on Delivery) payment method (default) to place the order.

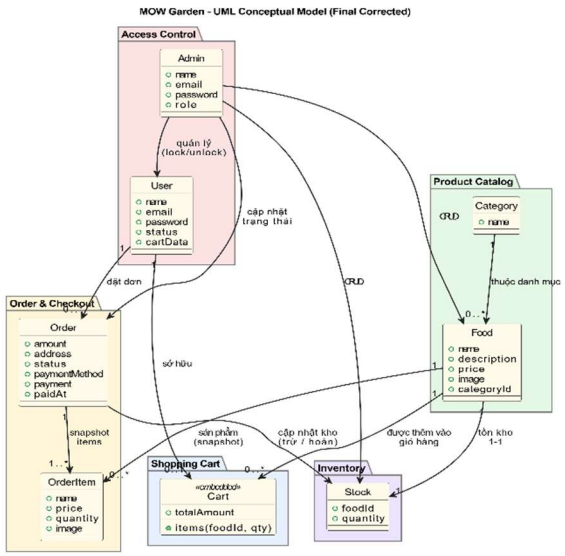
2.1.2.2 Order screen

Pic 15 Order screen

This screen displays the user's order details after they have successfully placed an order.

2.1.3 Data design

2.1.3.1 Conceptual Model



Pic 16. Conceptual Model

The Conceptual Model of MOW Garden describes the core entities of the system and how they are interconnected according to the plant retail business domain. The **User** entity is the center of all activities: each user owns a single shopping cart and can create multiple orders throughout their usage of the system. Products are organized into categories through the **Category** and **Food** entities, where each category can contain multiple products. When a user selects products for purchase,

the information is temporarily stored in the **Cart** and later transformed into an **Order**, which consists of multiple **OrderItem** records representing snapshots of products at the time of purchase.

Inventory is managed through the **Stock** entity, where each product has exactly one corresponding stock record, ensuring data consistency between the product catalog and inventory management. Orders directly affect inventory by deducting quantities when an order is placed and restoring quantities when an order is canceled. In addition, **Admin** plays a supervisory role by managing products, controlling inventory, updating order statuses, and locking or unlocking user accounts. Overall, the model clearly illustrates the business workflow: user login → product browsing → add to cart → COD checkout → inventory update → admin monitoring, ensuring that all operations are tightly connected and consistent.

Entity Relationships in the Conceptual Model

The Conceptual Model clearly illustrates the relationships among the main entities of the system. Each relationship arrow represents a specific business meaning based on how the system actually operates.

(1) User – Cart Relationship (1 → 0..1)
A user can have at most one shopping cart, which is stored directly within their account (cartData). When no products have been added, the cart may be empty; therefore, the relationship is **1 User – 0 or 1 Cart**.

(2) User – Order Relationship (1 → 0..*)
A user may place multiple orders or may have never placed any order at all. Therefore, the relationship between User and Order is **1 User – many Orders**.

(3) Category – Food Relationship (1 → 0..*)
Each category can contain multiple products, but a category is still valid even if it currently contains no products. Each Food item belongs to exactly one Category.

(4) Food – Cart Relationship (1 → 0..*)
A product can be added to the shopping carts of multiple users, or it may not be added by anyone. Therefore, Food is related to Cart as **1 product – many cart entries (across multiple users)**.

(5) Food – OrderItem Relationship (1 → 0..*)
During checkout, each OrderItem represents a snapshot of a Food item at the time of purchase. A single Food item may appear multiple times across different orders. Hence, the relationship is **1 Food – many OrderItems**.

(6) Order – OrderItem Relationship (1 → 1..*)
An order must contain at least one product and may contain multiple products. Therefore, the relationship between Order and OrderItem is **1 Order – 1 or more OrderItems**.

(7) Food – Stock Relationship (1 → 1)
Each Food product has exactly one corresponding Stock record, and each Stock record belongs to exactly one Food product. This represents a clear **one-to-one (1–1)** relationship within the system.

(8) Order – Stock Relationship (Update Interaction)
When an order is created, the system deducts inventory quantities for the corresponding products; when an order is canceled, inventory quantities are restored. This is not a static data relationship but a **business interaction** between Order and Stock.

(9) Admin → User / Food / Stock / Order Relationship
Admin is not an inherited entity of User but represents a completely separate group of users. Admin has the authority to:

- Lock or unlock User accounts (many Users managed by one Admin)

- Perform CRUD operations on products (Food)
- Manage inventory (Stock)
- Update order statuses (Order)

This is a **management relationship**, not a direct data relationship.

2.1.3.2 Data Model & Business Rules

The data model of MOW Garden is designed based on domain analysis and the relationships among business components. The system consists of five main data groups: users and access control, categories and products, shopping cart, orders, and inventory. Each group is separated into individual collections in MongoDB and linked using ObjectId references or embedded documents, depending on the nature of the data. Data tightly coupled to a single entity, such as a user's shopping cart or delivery information within an order, is stored as embedded documents to optimize query performance and avoid complex join operations. Meanwhile, independent data such as products, categories, and inventory is organized into separate collections to ensure scalability and ease of management.

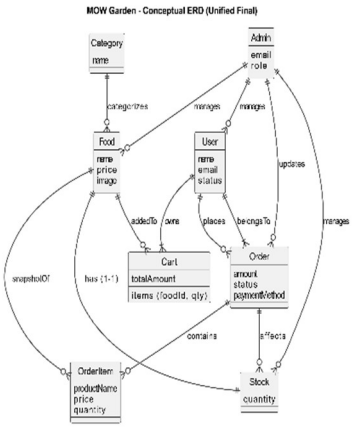
In parallel with the data model, the system enforces a set of business rules to maintain consistency throughout its operation. Users must have valid accounts to perform any actions; locked accounts are not allowed to log in or create orders. When an administrator adds a new product, a corresponding inventory record must be created immediately with an initial quantity of zero; when a product is deleted, its associated inventory record must also be removed to prevent redundant data. The user's shopping cart functions as a temporary storage area, where any change in product quantity is directly updated in the user's cartData.

During the order placement process, the system must verify inventory availability before creating an order to prevent insufficient stock scenarios. If stock levels are sufficient, the cart information is recorded as snapshots in OrderItem to ensure that purchase history remains unaffected by future administrative changes to product data. Inventory deduction occurs immediately when an order is created, while inventory restoration is performed only when the order is canceled. For successfully delivered orders, the system automatically marks the COD payment as completed. Administrators are the only actors authorized to update order statuses and directly affect inventory data through order-related operations.

The document-oriented data structure, combined with a strict set of business rules, enables the system to maintain stability, prevent data inconsistencies, and accurately

reflect the full processing flow—from user login and product selection to order placement, confirmation, and inventory updates.

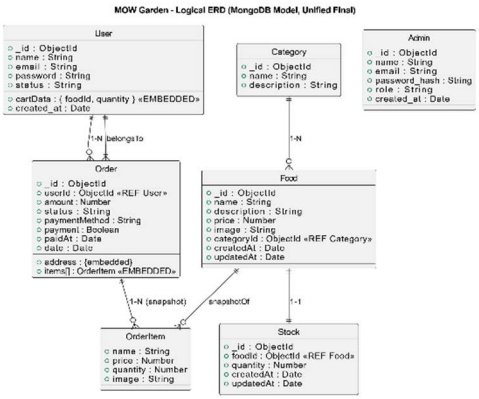
2.1.3.2 Conceptual ERD



Pic 17. Conceptual ERD

The Conceptual ERD diagram illustrates the core business entities of the MOW Garden system and how they are interconnected throughout the sales process. Users own a single shopping cart and can place multiple orders, while administrators play a coordinating role by managing products, orders, inventory, and user account statuses. Each product belongs to a specific category and has exactly one corresponding inventory record. Both shopping carts and orders contain lists of products, where OrderItem represents a snapshot of product information at the time an order is placed. At this level, the diagram focuses solely on business relationships without detailing physical data storage, providing a clear overview of the interaction flow between users, product data, and order processing activities.

2.1.3.3 Logical ERD



Pic 18. Logical ERD

The Logical ERD illustrates how the system is practically implemented in MongoDB. Core entities such as **User**, **Admin**, **Category**, **Food**, **Stock**, and **Order** are separated into independent collections and are linked to each other using ObjectId references. Dependent data is embedded directly within parent documents; for example, the shopping cart is embedded within the User document, and the list of OrderItem records is embedded within the Order document to accurately reflect the document-oriented nature of MongoDB.

Each product is linked to its category via categoryId and has exactly one corresponding inventory record in the Stock collection. The Order collection stores complete delivery information along with a snapshot of purchased products. The Logical ERD clearly demonstrates the actual data storage structure of the system and how the collections collaborate to support the business workflow.

2.1.3.4 Data Dictionary

Collection: Order

Attribute	Type	Description
Attribute	**Type**	**Description**
firstName	String	Recipient's first name
lastName	String	Recipient's last name
email	String	Recipient's email address
street	String	Delivery street address
city	String	City
state	String	District / Province
zipcode	String	Postal code
country	String	Country
phone	String	Contact phone number

Embedded Object: address

Attribute	Type	Description
firstName	String	Recipient's first name
lastName	String	Recipient's last name

email	String	Recipient's email address
street	String	Delivery street address
city	String	City
state	String	District / Province
zipcode	String	Postal code
country	String	Country
phone	String	Contact phone number

The design of the **Order** collection uses embedded documents for delivery information and the list of purchased products to ensure historical data consistency. Storing product information as snapshots prevents orders from being affected by subsequent changes to product data, which aligns well with MongoDB's document-oriented architecture and common e-commerce business requirements.

2.2 Architecture Design

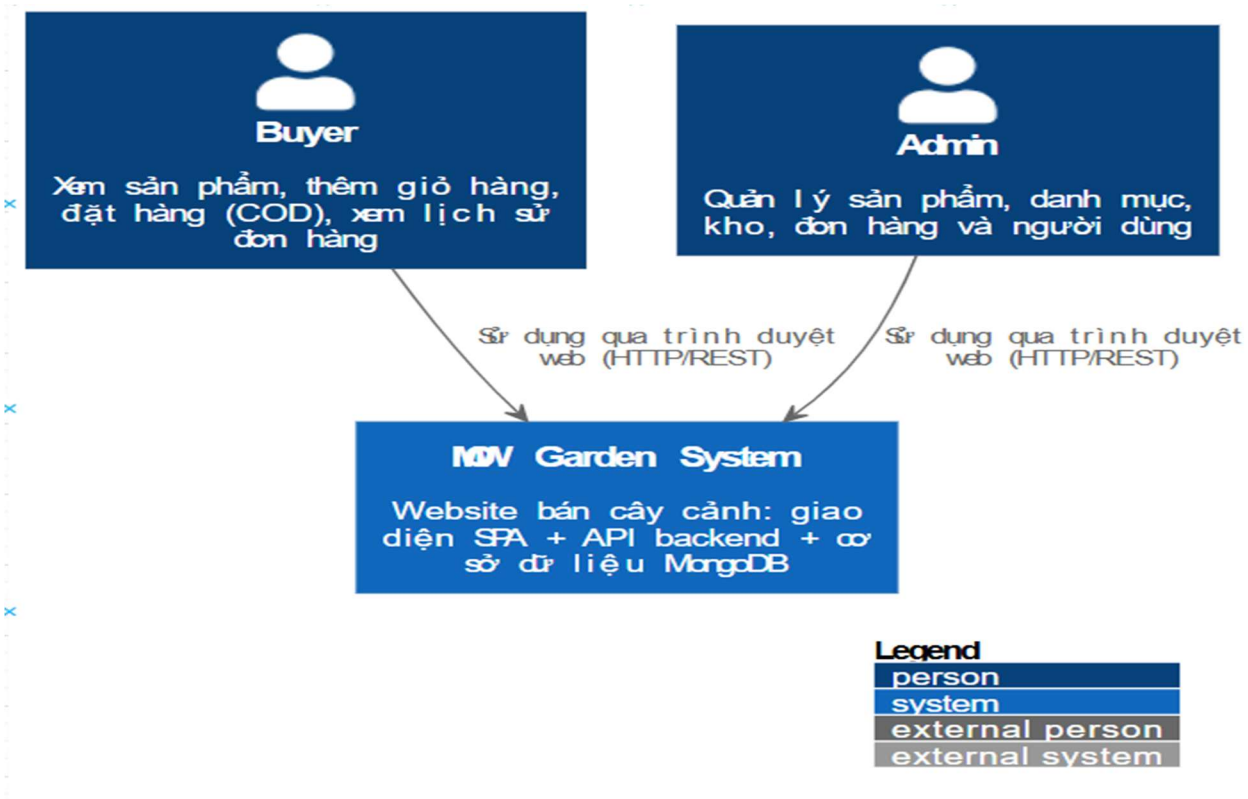
This section presents the system architecture of the MOW Garden Website based on the **C4 Architecture Model**, which describes the system from a high-level overview to detailed implementation perspectives.

The analysis is divided into four levels (C1 → C4), each corresponding to a specific architectural view:

View	Name	Main Focus
C1	System Context	Identifies the system context and interacting actors
C2	Container View	Analyzes the interactions between frontend, backend, and database
C3	Component View	Analyzes business modules and data flow
C4	Code / Implementation (Decomposition) View	Analyzes the actual implementation and source code structure

Decomposition View: C4

C1 – SystemContext



Pic 19. C1 – SystemContext

The C1 diagram provides the highest-level overview of the MOW Garden system and the actors that interact directly with it. The system operates as an online plant retail website, consisting of a single-page application (SPA) user interface and a backend API that handles all business logic, with data stored in MongoDB.

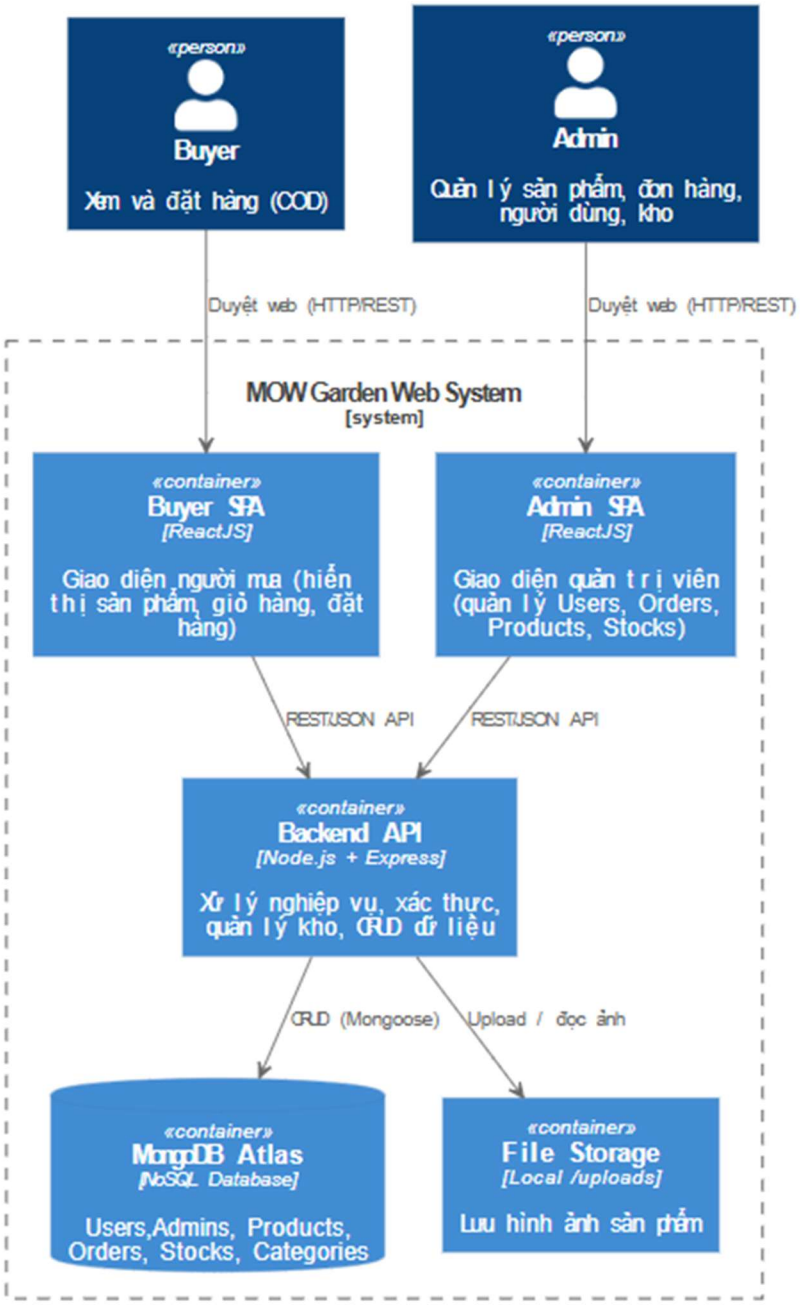
On the end-user side, **Buyers** access the system via a web browser to browse plant catalogs, add products to the shopping cart, place orders using the COD payment method, and track their purchase history. All user actions are sent to the system through HTTP/REST requests.

Administrators (Admin) also access the system via a web browser but in a management role. Admins log in to a dedicated administrative interface to manage products, categories, inventory, orders, and user accounts. Admin interactions are also performed through the same API layer as Buyers, but with elevated privileges.

At the center of the diagram is the **MOW Garden System**, which receives requests from both Buyers and Admins, processes business logic related to purchasing and inventory updates, and persists all data. The C1 diagram clearly

identifies who uses the system, through which channels they access it, and the primary purposes of each user group.

C2-Container



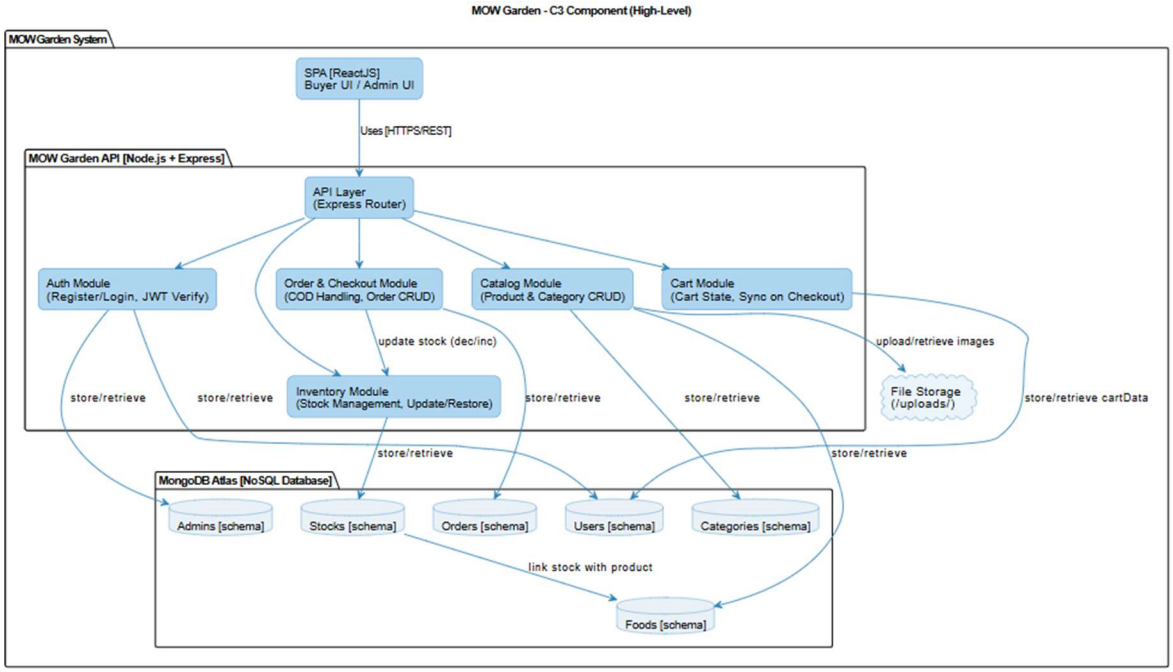
Pic 20. C2-Container

The C2 diagram illustrates how the MOW Garden system is decomposed into major containers and how they interact during runtime. At the presentation layer, both Buyers and Administrators access the system through a web browser, but they are separated into two distinct ReactJS applications: **Buyer SPA** and **Admin SPA**. The Buyer SPA supports product browsing, shopping cart operations, and COD order placement, while the Admin SPA is used to manage all system data, including users, orders, inventory, and categories.

Both SPAs communicate directly with the Backend API via RESTful APIs. The backend is built using **Node.js** and **Express**, serving as the core business logic layer responsible for user authentication, inventory updates, and all CRUD operations across the system. The backend uses **Mongoose** to connect to **MongoDB Atlas**, which stores all primary data collections such as Users, Admins, Products, Orders, Stocks, and Categories. In addition, the backend manages a dedicated file storage directory for product images, while only image metadata is stored in the database to optimize storage usage.

The main system flow is straightforward: the SPA sends requests → the backend processes business logic → queries MongoDB or reads/writes image files → returns responses to the frontend. This clear separation of containers enhances scalability, simplifies deployment, and ensures that each component has a well-defined responsibility with minimal coupling.

C3-Component(High-Level)



Pic 21. C3-Component(High-Level)

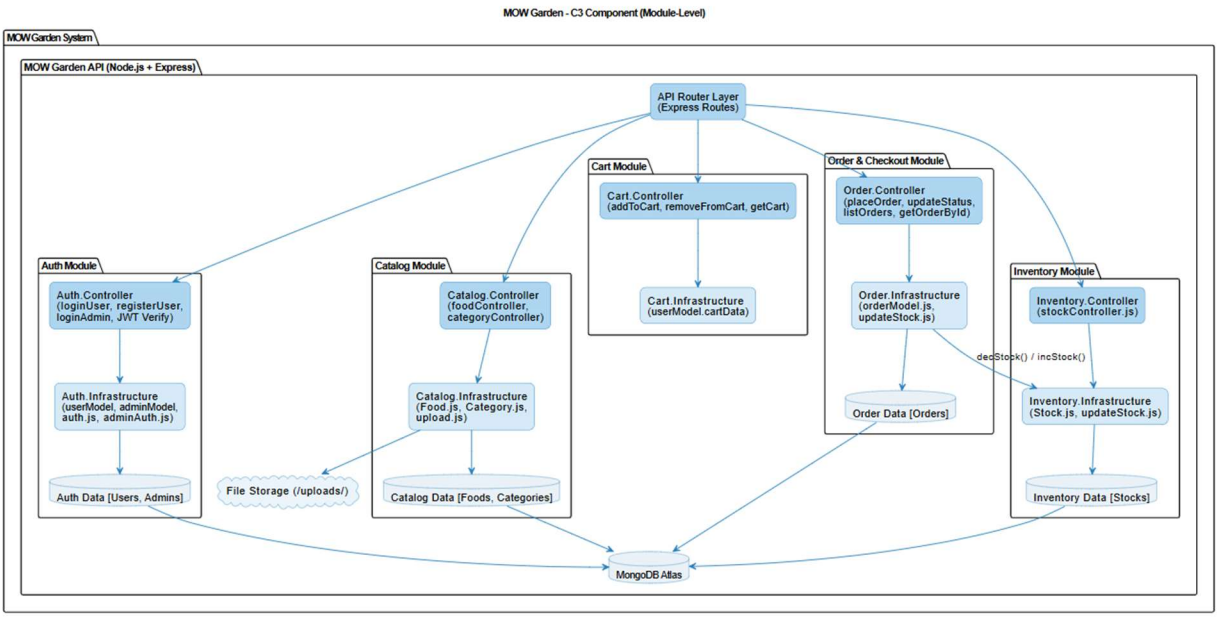
The C3 diagram describes how the entire MOW Garden system is decomposed into functional components and how these components collaborate during system operation. At the presentation layer, two separate React-based applications serve two user groups: Buyers use the interface to browse products, manage the shopping cart, and place COD orders, while Administrators use the management interface to update products, categories, orders, and inventory. Both interfaces are responsible only for rendering the UI and sending HTTP requests to the backend.

The core processing layer of the system is the Backend API, implemented using Node.js and Express. Incoming requests are routed through controllers and forwarded to the appropriate business modules. Each module handles a specific responsibility: the authentication module manages user registration and login using JWT; the product module manages plant and category data; the cart module maintains shopping cart data for each user; the order module processes COD orders and delivery statuses; and the inventory module is responsible for updating and restoring stock quantities when orders are created or canceled. These modules communicate with each other only through data persisted in MongoDB Atlas.

All primary data is stored as collections in MongoDB, including Users, Admins, Foods, Categories, Orders, and Stocks. Product images are not stored in the database; instead, they are saved in an uploads directory, with only the file names stored as metadata in the database. When the frontend sends a request to update product data, the backend processes images using multer, writes the files to disk, and stores the corresponding metadata in the database.

The overall system flow is largely linear: users interact with the UI → the UI sends requests → the backend routes the requests → business modules process logic and query data → the backend returns JSON responses to the UI. By clearly separating the backend into well-defined modules, the architecture remains consistent, easy to understand, and highly extensible for future development.

C3-Component(Model Level)



Pic 22. C3-Component(Model Level)

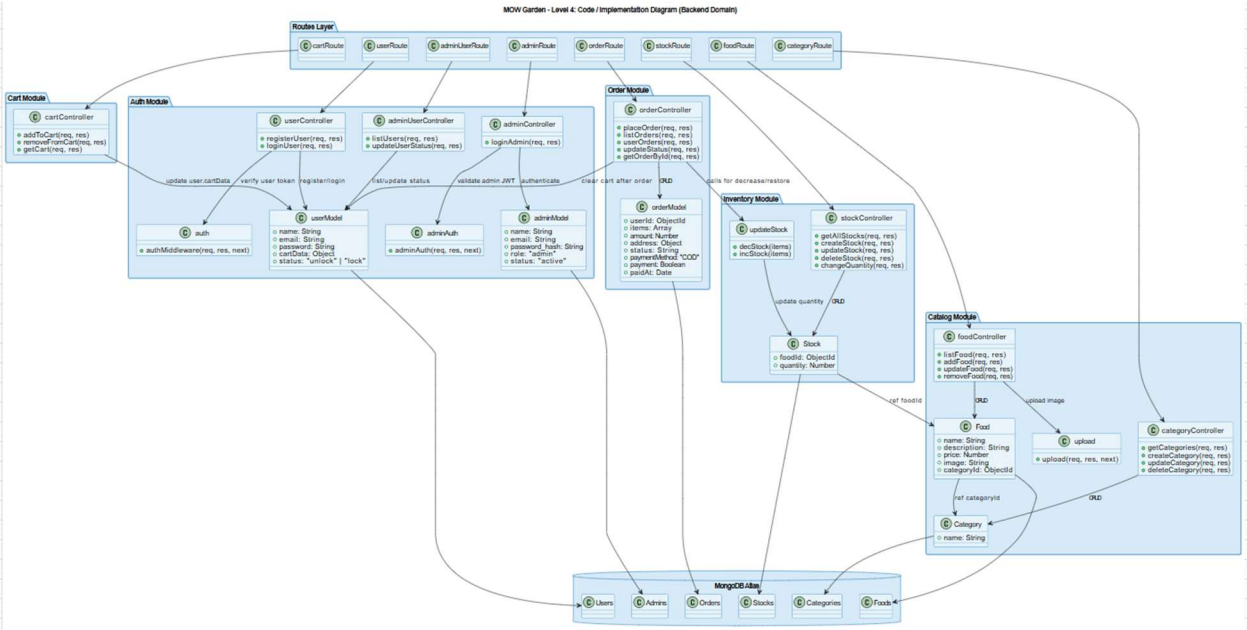
The C3 diagram describes how the entire MOW Garden system is decomposed into functional components and how these components collaborate during system operation. At the presentation layer, two separate React-based applications serve two user groups: Buyers use the interface to browse products, manage the shopping cart, and place COD orders, while Administrators use the management interface to update products, categories, orders, and inventory. Both interfaces are responsible only for rendering the UI and sending HTTP requests to the backend.

The core processing layer of the system is the Backend API, implemented using Node.js and Express. Incoming requests are routed through controllers and forwarded to the appropriate business modules. Each module handles a specific responsibility: the authentication module manages user registration and login using JWT; the product module manages plant and category data; the cart module maintains shopping cart data for each user; the order module processes COD orders and delivery statuses; and the inventory module is responsible for updating and restoring stock quantities when orders are created or canceled. These modules communicate with each other only through data persisted in MongoDB Atlas.

All primary data is stored as collections in MongoDB, including Users, Admins, Foods, Categories, Orders, and Stocks. Product images are not stored in the database; instead, they are saved in an uploads directory, with only the file names stored as metadata in the database. When the frontend sends a request to update product data, the backend processes images using multer, writes the files to disk, and stores the corresponding metadata in the database.

The overall system flow is largely linear: users interact with the UI → the UI sends requests → the backend routes the requests → business modules process logic and query data → the backend returns JSON responses to the UI. By clearly separating the backend into well-defined modules, the architecture remains consistent, easy to understand, and highly extensible for future development.

C4- Code/Implementation Diagram:



Pic 23. C4- Code/Implementation Diagram

The C4 diagram describes how the entire MOW Garden backend is organized in the actual source code. Components at Level 3 are decomposed into controllers, models, middleware, and routes, clearly illustrating the data processing flow from request handling to data persistence in MongoDB or image file storage.

The backend is divided into five main business modules, each following a three-layer structure: **Route → Controller → Model**, with additional middleware applied to APIs that require authentication.

The **Auth Module** manages Buyer login, Admin login, and access control. Its controllers handle credential validation, JWT token generation, and blocking locked users. Two middleware components, auth.js and adminAuth.js, ensure that only valid Buyers or authorized Admins can access the corresponding routes. This module interacts directly with the **Users** and **Admins** collections.

The **Catalog Module** is responsible for products and categories. When an admin creates or updates a product, the controller also handles image uploads via upload.js, stores image metadata in the **Food** collection, and simultaneously creates or manages the corresponding inventory record in **Stock**. This is the only module that interacts directly with the file storage layer to read and write actual image files.

The **Cart Module** stores the shopping cart directly within the **User** document. Controllers operate only on the cartData field to increase, decrease, or return the current cart state. There is no separate cart collection; embedding the cart within the User document ensures that cart data persists across page reloads.

The **Order Module** is responsible for the COD order workflow, including inventory validation, stock deduction, order creation, cart clearing, and order status management. When an Admin updates an order status, corresponding inventory changes are triggered, such as stock restoration or payment finalization. All order-related data is stored as snapshots within the **Order** and **OrderItem** structures.

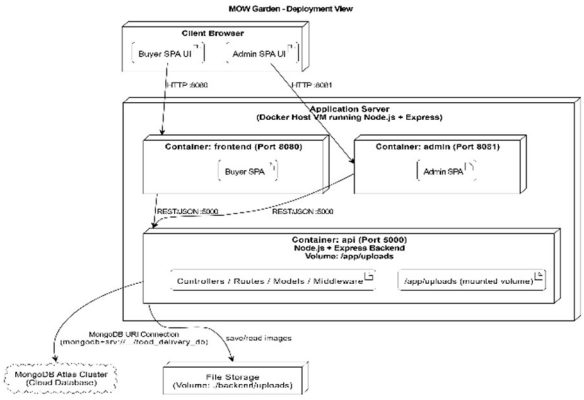
The **Inventory Module** manages actual stock quantities. Inventory can be increased, adjusted, or removed independently. Two core functions, decStock() and incStock(), act as bridges between the Order and Inventory modules, ensuring that every purchase or cancellation operation results in accurate stock updates.

The **Routes layer** serves as the single entry point of the backend. Each route file maps API endpoints to the appropriate controller and applies middleware to protect sensitive APIs. The **Mongoose models** connect the application to MongoDB Atlas,

with each model corresponding to a specific collection: **Users**, **Admins**, **Foods**, **Categories**, **Stocks**, and **Orders**.

Overall, the C4 diagram demonstrates a clearly modularized Node.js backend, where data flows linearly through **Route → Controller → Model → MongoDB**, with an additional file storage layer for image handling. This organization enhances code readability, scalability, and accurately reflects the business workflow of the MOW Garden plant shop.

Deployment view



Pic 24. Deployment view

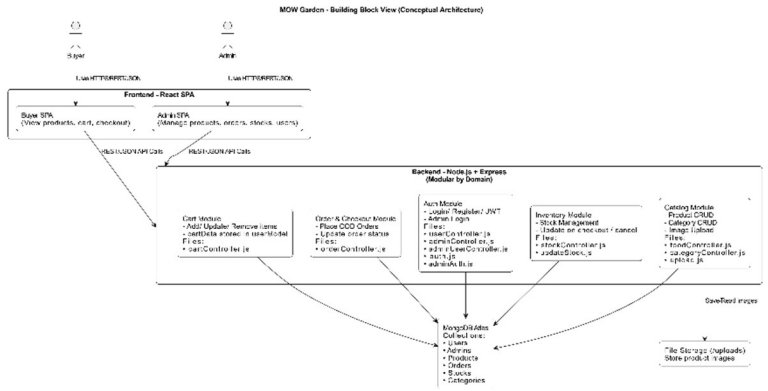
The Deployment View diagram describes how the MOW Garden system is deployed in a real-world environment. Users access the system through a web browser, interacting with two separate user interfaces: the Buyer SPA and the Admin SPA. These two interfaces run independently in separate frontend and admin containers (ports 8080 and 8081) and are built into static files that are served directly to the browser. All user interactions from the frontend are sent to the Backend API via REST/JSON through port 5000.

The Backend API runs in a dedicated container, using Node.js and Express to handle all business logic such as authentication, product management, order processing, inventory management, and shopping cart operations. This container mounts a volume at /app/uploads to store product images, while image metadata is stored within MongoDB documents.

All system data is stored entirely on MongoDB Atlas, connected via a mongodb+srv://... connection string; therefore, the server does not require an

internal database container. In addition, actual product images are stored in the backend/uploads directory on the host machine through a volume mounted to the API container. This separation of components simplifies deployment, improves scalability, and ensures that each part of the system operates independently and fulfills its designated responsibility.

Block Diagram



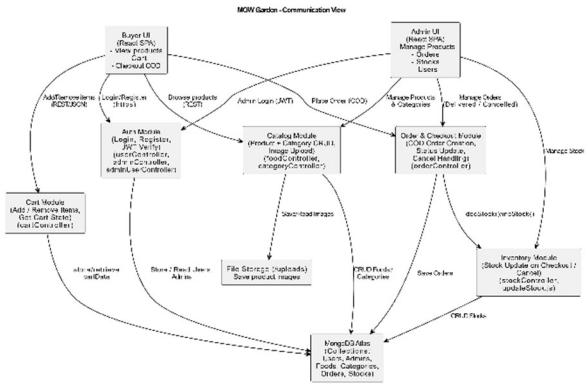
Pic 25. Block Diagram

The **Building Block View** describes how the MOW Garden system is composed of its main functional building blocks. On the presentation side, the system consists of two separate SPAs: the **Buyer SPA**, which serves customers, and the **Admin SPA**, which is dedicated to administrative users. All interactions on the user interfaces are forwarded to the backend through REST APIs.

The backend is organized into distinct business modules, including authentication, product–category management, shopping cart, COD order processing, and inventory management. Each module contains its own controllers, middleware, and direct connections to the MongoDB Atlas database. Product images are handled separately and stored in the server's uploads directory.

The diagram clearly illustrates the end-to-end flow from **user interface → API → business modules → data storage**, while also highlighting a **monolithic backend architecture** that is cleanly modularized by domain responsibilities.

CommunicationView



Pic 26. CommunicationView

The **Communication View** illustrates how Buyers and Admins send requests from the React SPA interfaces to the backend system. Buyers primarily interact with the **Catalog**, **Cart**, and **Order** modules to browse plants, add items to the shopping cart, and place COD orders. Admins perform more extensive operations, including managing products, orders, and inventory.

The backend receives requests through the corresponding business modules, then reads from and writes to **MongoDB Atlas**, while product images are stored in the server's uploads directory. The **Order Module** and **Inventory Module** are tightly coordinated to deduct or restore stock quantities based on order status changes.

This diagram focuses on clearly describing where requests originate, where they are processed, and which modules handle the data—accurately reflecting how the MOW Garden system operates in the actual codebase.

CHAPTER 3: Test Plan

3.1. Introduction

3.1.1. Purpose

The purpose of this document is to define the testing approach for the **MOW Garden Website** project. It helps the project team clearly understand the test scope, organization, acceptance criteria, potential risks, and the resources required.

The main contents include:

- An overview of the project and the testing scope.
- A list of requirements and acceptance criteria.
- A detailed testing strategy and test plan.
- Resources, test environment, and project milestones.
- Final deliverables.

3.1.2. Definitions, Acronyms, and Abbreviations

Abbreviation	Description
AT	Acceptance Test
IT	Integration Test
SRS	Software Requirement Specification
QA	Quality Assurance
DB	Database
UAT	User Acceptance Testing
UI	User Interface

3.1.3. References

Title/File name	Author	Effective Date
Business Context – MOW Garden	Project Team	2025-09-25
SRS – MOW Garden	Project Team	2025-09-27
Test Guideline Document	Fsoft	2024-12-01

3.1.4. Background information

MOW Garden is an e-commerce website for selling ornamental plants and gardening accessories. The system allows users to:

- Browse product categories, and search and filter products by type or price.

- View detailed product information (images, descriptions, and prices).
- Add or remove products from the shopping cart.
- Place orders (checkout).
- Receive notifications when an order is successfully placed.
- Administrators (Admin) manage inventory and confirm customer orders.

The objective of testing is to ensure that the system operates correctly, reliably, and securely, while delivering a positive user experience.

3.1.5. Scope of testing

In Scope:

- Functional testing (catalog, cart, checkout, inventory, notifications, and access control).
- User Interface testing (UI/UX).
- Database testing (data consistency and no data loss).
- Basic performance testing (small to medium concurrent load).

Out of Scope:

- International payment processing.
- Integration with external logistics systems.
- Advanced non-functional requirements (enterprise-level scalability).

3.1.6. Constraints

- The testing period is limited to two weeks.
- Human resources are limited (4 team members).
- The system is currently deployed only in a local environment and a demo server; a full production environment is not yet available.

3.1.7. Risk List

Risk	Impact	Mitigation
Lack of a production environment	Test results may not fully reflect real-world conditions	Use mock data and a staging server
Short testing timeframe	Some test cases may be missed	Prioritize critical functional test cases

Limited human resources	Potential delays in testing progress	Assign clear roles and focus on high-priority tasks
Lack of automation tools	Heavy reliance on manual testing	Use basic tools such as Postman and JMeter

3.1.8. Training Needs

The team requires short-term training on the following topics:

- Writing and managing test cases.
- Using Postman for API testing.
- Using JMeter for basic performance testing.
- Bug reporting and issue tracking using GitHub.

3.2. Requirements for Testing

3.2.1. Test Items

The following list identifies the main features and functions selected as testing targets for the **MOW Garden** project:

No.	Name of Function	Outline of Functions	Notes
1	Product Catalog	View product lists, search and filter by category and price. View product details (name, description, price, images).	Includes UI testing and search/filter functionality
2	Shopping Cart	Add/remove products to/from the cart, update quantities, calculate total price.	Tested in both logged-in and guest scenarios
3	Checkout Process	Enter order information, select payment method, and confirm orders.	Includes data validation
4	Inventory Management	Manage stock quantities and update inventory upon order placement.	Admin-side functionality
5	Access Control (User/Admin)	User registration, login, and role-based authorization between User and Admin.	Includes authentication and basic security testing

No.	Name of Function	Outline of Functions	Notes
6	Order Management	View order lists, view order details, and track order status (User & Admin).	Includes data display and order status updates
7	Notification Service	Send notifications when an order is successfully placed.	Push notification testing
8	Product Management	Create, read, update, and delete products and link them to inventory.	Admin authorization applied

3.2.2. Acceptance Test Criteria

To be accepted, the software must meet the following testing criteria:

- **Test coverage** ≥ 90%
- **Successful test coverage** ≥ 95%
- All defined test cases (Unit / Integration / System Tests) must be fully executed
- The number of defects or weighted defects must be ≤ 5 per main module

3.3. Features to Be Tested

The following list defines the items—including use cases, functional requirements, and non-functional requirements—selected as testing targets. These items represent the key components that need to be tested in the **MOW Garden** system.

3.3.1. Functionality

3.3.1.1. Product Catalog

- Verify that the system can correctly display the complete list of products retrieved from the database.

- Verify that users can filter products by category, price, or stock status.

- Verify that users can search for products using keywords.

- Verify that the product list is automatically updated when inventory changes occur.

3.3.1.2. Product Details

- Verify that the system displays accurate product information, including name, description, price, images, and remaining quantity.

- Verify that users can view multiple images of the same product.

- Verify that out-of-stock products are clearly indicated and cannot be added to the cart.

- Verify that when an Admin updates product information, the product detail page is updated automatically.

3.3.1.3. Shopping Cart

- Verify that users can add products to the cart from the product list or product detail page.

- Verify that users can update product quantities in the cart.

- Verify that users can remove products from the cart.

- Verify that the cart total price is calculated correctly and displayed in the correct format.

- Verify that if a product is removed or becomes out of stock, the cart status is automatically updated.

- Verify that the system correctly handles adding the same product multiple times.

3.3.1.4. Checkout Process

- Verify that users can enter shipping and payment information.

- Verify that the system validates all required input data (name, phone number, address).

- Verify that orders are stored in the database and displayed in the order history.

- Verify that users receive notifications after successful order placement.

- Verify that inventory quantities are reduced correctly after checkout.

- Verify that order status is updated after payment (e.g., “Confirmed”, “Shipping”).

3.3.1.5. Inventory Management

- Verify that inventory quantities are automatically updated after each order.

- Verify that out-of-stock products cannot be added to the cart.

- Verify that Admin users can update product information (price, stock, description).

- Verify that inventory is updated correctly when orders are completed or canceled.

- Verify that inventory changes are reflected in real time on the product detail page.

3.3.1.6. Access Control

- Verify that the system correctly enforces authorization between buyers and admins.

- Verify that users can only perform actions appropriate to their assigned roles.

- Verify that Admin users can add, edit, and delete products, while buyers can only purchase products.

- Verify that users can update their personal information.

- Verify that the “Forgot Password” functionality works correctly.

- Verify that Admin users can lock or unlock user accounts.

- Verify that users can update personal details such as name, address, and phone number.

3.3.1.7. Notification Service

- Verify that users receive notifications after successful order placement.

- Verify that Admin users can send promotional or order status notifications to users.

- Verify that notifications are displayed correctly on the user interface.

3.3.1.8. Order Management

- Verify that users can view their order list, including order ID, creation date, total amount, and status.

- Verify that users can view detailed information for each order (shipping info, items, invoice, status).

- Verify that order status is displayed correctly according to progress (Pending → Processing → Shipping → Completed).

- Verify that Admin users can view all customer orders.

- Verify that Admin users can update order status (e.g., from “Processing” to “Completed”).

- Verify that users receive notifications when Admin updates order status.

3.3.1.9. Product Management

- Verify that Admin users can add new products with complete information (name, description, price, images, stock).

- Verify that Admin users can edit product information.

- Verify that Admin users can delete products from the catalog.

- Verify that Admin users can assign products to inventory.

- Verify that the system prevents duplicate product names or IDs.

- Verify that when Admin users update or delete products, related data in **Product Catalog** and **Inventory** is updated accordingly.

3.3.2. Usability

3.3.2.1. Session Support

- Verify that users remain logged in throughout the working session.

- Verify that cart data is preserved when users reload the page or temporarily leave the site.

- Verify that the system automatically logs users out after a defined period of inactivity for security purposes.

- Verify that when users manually log out, all temporary session data is securely cleared.

3.3.2.2. Display Support

- Verify that the interface is displayed correctly on common browsers (Chrome, Edge, Safari, Firefox).

- Verify that the layout automatically adapts to different screen sizes (desktop, tablet, mobile).

- Verify that fonts, colors, and icons are clearly displayed and do not change across different resolutions.

- Verify that the system supports dark mode or customizable UI settings (if applicable).

3.3.3. Design Constraints

3.3.3.1. Test Environment

- Verify that the system operates stably on the defined test environments (Windows 10/11, macOS).

- Verify that supported browser versions are clearly defined (e.g., Chrome 120+, Edge 110+).

- Verify that the backend (Node.js/Express) and frontend (ReactJS) are successfully deployed in the staging environment.

- Verify that the database system (MongoDB) operates normally and data is stored and retrieved accurately.

3.3.3.2. Character Support

- Verify that the system supports Vietnamese characters, special characters, currency symbols, and emojis.

- Verify that data containing special characters does not cause display or database errors.

- Verify that the system correctly handles Unicode input (e.g., names, addresses, product descriptions).

3.3.4. Interfaces

- Verify that page layouts (Home, Product List, Product Detail, Cart, Checkout, Admin Dashboard) are properly organized.

- Verify that all buttons, links, and UI elements respond correctly to user interactions (hover, click, focus).

- Verify that input forms display clear validation messages when data is missing or invalid.

- Verify that system notifications (toast messages, alerts) clearly indicate success or failure.

- Verify that the Admin interface allows intuitive management of products, orders, and inventory.

- Verify that backend APIs provide accurate data to the frontend, return valid JSON responses, and use correct HTTP status codes.

3.4. Features Not to Be Tested

The following list includes features, functions, or requirements that are **not included in the testing scope** of the current release. These features may be considered in future development phases or extended versions.

3.4.1. Functional Features

- Third-Party Online Payment Integration (Payment Gateway Integration):
Integration and transaction processing with payment services such as PayPal, MoMo, ZaloPay, or VNPay have not been implemented and are therefore not tested in this phase.

- Product Recommendation System:
Features that recommend products based on user behavior or purchase history are not included in the current testing scope.

- Advanced User Management Features:
Advanced user management functions such as password recovery, two-factor authentication (2FA), and social login (Google, Facebook) have not been implemented and are not tested in this phase.

3.4.2. Non-Functional Features

- Advanced Performance Testing (Stress Testing):
Performance testing under extremely high concurrent user loads (over 10,000 simultaneous users) is not conducted in this phase and will be considered in future stages.

- Scalability Testing:
Scalability tests involving server clustering, load balancing, or enterprise-level infrastructure are outside the current testing scope.

- Advanced Security Testing:
In-depth security testing such as penetration testing or advanced injection attack simulations is not performed in this phase.

3.5.1.1. Functional Testing

Functional Testing aims to ensure that all system functions operate correctly according to the specified requirements.
For the **MOW Garden** system, this testing focuses on core functionalities such as product browsing, searching, adding items to the cart, checkout, payment, account management, and inventory management.

Item	Description
Test Objective	Verify that core system functionalities (view products, search, add to cart, checkout, account management, inventory management) work correctly according to business requirements.
Technique	- Execute test cases based on use cases and functional requirements - Use valid and invalid test data - Verify input validation and business rules - Tools: Postman, Selenium, JUnit
Completion Criteria	- All functional test cases are executed - Critical defects are fixed - No blocking defects remain
Special Considerations	Special focus on critical flows such as adding to cart without login, canceling orders during processing, and temporary cart storage

3.5.1.2. Business Cycle Testing

Business Cycle Testing ensures that the entire business workflow operates smoothly from start to end without interruption.
In **MOW Garden**, this includes the flow: product selection → add to cart → checkout → inventory update → order management.

Item	Description
Test Objective	Ensure that core business processes (purchase → payment → inventory update) operate smoothly

Item	Description
Technique	- Perform end-to-end testing - Verify data flow across modules (Product → Cart → Order → Inventory)
Completion Criteria	All core business workflows are completed successfully without errors
Special Considerations	Simulate real scenarios such as order cancellation, payment failure, and out-of-stock situations

3.5.1.3. User Interface Testing

User Interface Testing ensures that the web interface is displayed correctly, is easy to use, and responds accurately to user interactions.

Item	Description
Test Objective	Ensure the UI is displayed correctly, responsive, and user-friendly
Technique	Cross-browser testing (Chrome, Firefox, Edge, Safari), responsive testing (desktop, tablet, mobile), exploratory testing, Selenium
Completion Criteria	No Major or Blocker UI defects; at least 95% of target pages pass responsive testing
Special Considerations	Test under slow network conditions, large image loading, and font fallback behavior

3.5.1.4. Security and Access Control Testing

This testing ensures that the system is secure, only authorized users can access protected resources, and user data is protected.

Item	Description
Test Objective	Verify system security, access control, and protection against attacks
Technique	- Test login/logout and session timeout - Check SQL Injection, XSS, CSRF - Verify admin/user authorization
Completion Criteria	- No authentication bypass - No critical vulnerabilities from OWASP Top 10
Special Considerations	Verify SSL/HTTPS usage and password encryption

3.5.1.5. Performance and Load Testing

Performance and Load Testing evaluates whether the system meets performance and stability requirements under concurrent access.

Item	Description
Test Objective	Ensure the system operates stably under high load (many concurrent users and products)
Technique	- Use JMeter or Locust to simulate 1,000+ concurrent users - Perform Load Test, Stress Test, and Volume Test
Completion Criteria	- Average response time < 3 seconds with 500 concurrent users - System does not crash under load
Special Considerations	Define specific SLA thresholds (response time, maximum concurrent users)

3.5.1.6. Regression Testing

Regression Testing ensures that existing functionalities continue to work correctly after new features are added or bugs are fixed.

Item	Description
Test Objective	Verify that existing functionalities are not affected by changes
Technique	- Re-run previously passed test cases - Use automation tools (Selenium, Jest)
Completion Criteria	- All existing test cases pass - No unexpected defects are introduced
Special Considerations	Integrate regression testing into the CI/CD pipeline

3.5.1.7. Data and Database Integrity Testing

This testing ensures that data is not lost, duplicated, or corrupted during system operations.

Item	Description
Test Objective	Ensure product, cart, order, and user data is stored accurately and consistently
Technique	- Perform CRUD operations on the database - Verify transactions during checkout - Validate primary and foreign key constraints - Use SQL scripts and automation
Completion Criteria	- No data loss or duplication - Data integrity maintained across database tables
Special Considerations	Verify rollback behavior when errors occur

3.5.2. Test Stages

Different types of testing are executed at different stages of the testing lifecycle:

- **Unit Test:** Testing individual functions or modules, usually performed by developers.
- **Integration Test:** Testing interactions between multiple modules or APIs.
- **System Test:** Testing the entire system as a complete product.
- **Acceptance Test:** Validating that the system meets business requirements.

Type of Test	Unit	Integration	System	Acceptance
Functional Test	✓	✓	✓	✓
Business Cycle Test		✓	✓	✓
User Interface Test		✓	✓	✓
Data & Database Integrity Test	✓	✓	✓	
Performance Test			✓	
Security & Access Control Test		✓	✓	
Regression Test	✓	✓	✓	

- **Functional Testing:** Applied at all stages from Unit to Acceptance.
- **Business Cycle Testing:** Mainly executed from Integration stage onward.
- **UI Testing:** Starts from Integration (Frontend + Backend), then System and Acceptance stages.
- **Database Testing:** Begins at Unit level and continues through Integration and System stages.
- **Performance Testing:** Mainly executed at System Test stage.
- **Security Testing:** Starts from Integration stage (API, authorization, login).
- **Regression Testing:** Continuously executed from Unit → Integration → System to ensure changes do not break existing functionality.

3.5.3. Test Phases

Phase	Objective	Main Activities	Responsible	Deliverables
1. Unit Test	Verify that individual functions or small modules work correctly according to logic	- Write test code (Jest, Mocha) - Verify results of each function	Developer	- Unit Test Report - Tested source code
2. Integration Test	Ensure that modules (Catalog, Cart, Checkout, Inventory) are correctly connected and exchange data accurately	- Test API integration between Frontend and Backend - Test database transactions - Test API access permissions	Developer + Tester	- Integration Test Report - Defect Log
3. System Test	Verify the entire MOW Garden system as a complete product	- Execute comprehensive test cases (functional, UI, security, performance) - Test end-to-end workflow: Product → Cart → Checkout → Order → Inventory	Tester Team	- System Test Result - Defect Report - Test Summary
4. User Acceptance Test (UAT)	Ensure the system meets business requirements and is ready for end users	- Execute test cases based on user scenarios - Verify UI, business	Tester + Client Representative	- UAT Sign-off Document - Acceptance Report

Phase	Objective	Main Activities	Responsible	Deliverables
		processes, and system stability - Collect user feedback		
5. Regression Test	Ensure existing functionalities continue to work correctly after bug fixes or updates	- Re-run previously passed test cases - Perform regression automation using Selenium	Tester Team	- Regression Report - Updated Defect Log

3.6. Resource

3.6.1. Human Resource

Worker/Doer	Role	Specific Responsibilities	Location
Nguyễn Nhật Hải	Tester	Execute test cases, report defects	SGU – Ho Chi Minh City
Nguyễn Thành Việt	Tester	API testing, integration testing	SGU – Ho Chi Minh City
Nguyễn Thành Trung Hiếu	Tester	UI testing, regression testing	SGU – Ho Chi Minh City
Nguyễn Thành Đạt	Leader	Test planning, task assignment, reporting	SGU – Ho Chi Minh City

3.6.2. Test Management

Test Management Approach:

- Create test plans, assign tasks, and track progress using **Jira / Trello**.
- Conduct daily stand-up meetings to update testing progress.
- Submit weekly progress reports to the Project Manager (PM).

Defect Management:

- Record and track defects using **Jira**.
- Classify defects by severity level: **Critical, Major, Minor**.

Defect Handling Workflow:

Log defect → Assign to Developer → Fix defect → Re-test → Close defect

3.7. Test Environment

3.7.1. Hardware

Laptop/PC (for web testing):

- CPU: Intel Core i5 or higher
- RAM: 8GB or more
- Storage: SSD 256GB
- Quantity: 4 machines
- Usage duration: Entire testing period

Network Devices:

- Wi-Fi Router (IEEE 802.11ac standard)
- Internet connection speed: minimum 50 Mbps
- Quantity: 1 set

3.7.2. Software

Operating Systems:

- Windows 10 Pro (64-bit)
- Ubuntu 20.04 LTS

Web Browsers:

- Google Chrome (latest version)
- Mozilla Firefox (latest version)
- Microsoft Edge (latest version)

Testing Tools:

- Postman (API Testing)
- Selenium WebDriver (Automation Testing)
- JMeter (Performance Testing)

3.7.3. Infrastructure

Purpose	Tool	Vendor / In-house	Version
Defect logging	Jira	Atlassian	Cloud Version
Test case management	TestRail	Gurock Software	7.5
Task management	Trello	Atlassian	Cloud Version
Source code management	GitHub	GitHub Inc.	Cloud Version
Performance testing	JMeter	Apache	5.6
Time tracking (test effort)	Timesheet	In-house	1.2

3.8. Test Milestones

Test Phase	Activity Description	Planned Date
Test Planning	Complete the Test Plan, define test scope, objectives, resources, and test environment	01/10/2025 – 03/10/2025

Test Phase	Activity Description	Planned Date
Test Design	Analyze requirements, write test cases, prepare test data, and map test cases to user stories	04/10/2025 – 06/10/2025
Test Environment Setup	Set up the testing environment (frontend, backend, database, test accounts)	07/10/2025
Test Execution (System & Integration)	Execute Functional Testing, Business Flow Testing, UI Testing, and Security Testing. Record test results and log defects	08/10/2025 – 13/10/2025
Performance & Regression Testing	Execute performance testing, load testing, and regression testing after bug fixes	14/10/2025 – 15/10/2025
User Acceptance Testing (UAT)	Client/Product Owner validates system functionality based on business requirements	16/10/2025 – 17/10/2025
Test Closure & Reporting	Consolidate results, evaluate product quality, create Test Summary Report, and document Lessons Learned	18/10/2025

3.9. Deliverables

Deliverable	Description	Responsible
Test Plan Document	Overall test plan including scope, strategy, and schedule	Test Leader
Test Case Specification	Detailed test cases for each module (Catalog, Cart, Checkout, etc.)	Tester
Test Data Set	Test data used during testing (user data, sample products, sample orders)	Tester

Deliverable	Description	Responsible
Test Execution Log	Records of test execution status and results for each test case	Tester
Defect Report / Defect Log	List of defects identified during testing	Tester / Test Leader
Performance Report	Performance and load testing results (response time, throughput)	Tester
Test Summary Report	Overall testing results, pass/fail rates, and product quality assessment	Test Leader
UAT Sign-off Document	Formal acceptance confirmation from end users	Product Owner / Customer

CHAPTER 4: TEST DESIGN

4.1. Objectives of Test Design

Chapter 4 provides a detailed description of the test design activities for the MOW Garden Website system, based on the Test Strategy (Chapter 3) and the Software Requirements Specification (SRS).

The objectives of Test Design include:

- Defining how test cases, test scenarios, and test conditions are designed for each module.

- Ensuring that all testing activities comply with the defined scope, test types, and test stages specified in Chapter 3.

- Serving as a direct foundation for Test Case Specification (Chapter 5) and Test Execution.

4.2. Basis for Test Design

The Test Design is developed based on the following documents and inputs:

- Software Requirement Specification (SRS)
- Test Plan & Test Strategy (Chapter 3)
- Core business workflows of the system:

Access Control → Product Catalog → Shopping Cart → Checkout → Order → Inventory

4.3. Testing Methodologies and Techniques Applied to the Checkout Module

4.3.1. Black-box Testing Methodology

4.3.1.1. Overview

Black-box testing is a testing methodology in which the tester does not consider the internal code structure of the system. Instead, the focus is placed on:

- Input data
- System processing behavior

- Output results

For the Checkout module, black-box testing is applied as the primary testing methodology because this module is highly business-oriented and directly affected by user requirements and business rules.

All Checkout test cases (Chk-001 to Chk-013) are derived using black-box testing techniques.

4.3.1.2. Black-box Testing Techniques and Test Case Generation

(1) Use Case–based Testing

Technique Description

Use Case–based Testing designs test cases based on business use cases, simulating real user behavior.

Application to the Checkout Module

Main use case:

- **UC-Checkout**: User places an order using Cash on Delivery (COD)

Test Case Generation Approach

- Identify the main flow
- Identify alternative flows
- Identify exception flows

Generated Test Cases

- Chk-001: Successful checkout with valid information
- Chk-002: Missing required information
- Chk-009: Checkout with an empty cart

ID	Test Case Description	Test Case Procedure	Expected Output	Test data	Result	Description

Chk-001	Enter valid shipping information and place a COD order	1. Log in as a user 2. Add products to the cart 3. Navigate to the Checkout page 4. Enter valid name/phone/address 5. Select COD payment method 6. Click **"Confirm Order"**	Order is placed successfully; an order is created and the order ID is displayed	Valid information	Pass	Decision table/COD
Chk-002	Missing required shipping information	1. Navigate to the Checkout page 2. Leave the phone number or address empty 3. Click **Confirm** 4. Observe the error message	No order is created; an error message is displayed at the missing field	missing address/phone	Pass	Validation
Chk-009	Prevent checkout when the cart is empty	1. Open an empty cart 2. Access the Checkout page (if accessible) 3. Click **Confirm**	Checkout is blocked; the system prompts the user to add products	empty cart	Pass	Business rule

(2) Equivalence Partitioning

Technique Description

Equivalence Partitioning divides input data into equivalence classes, where the system is expected to behave similarly for all values within the same class.

Application to the Checkout Module

Input fields include:

- Phone number
- Shipping address
- Recipient name

Test Case Generation Approach

- Select representative values from valid equivalence classes
- Select representative values from invalid equivalence classes

Generated Test Cases

- Chk-001: Valid input data
- Chk-003: Invalid phone number format
- Chk-004: Invalid shipping address

ID	Test Case Description	Test Case Procedure	Expected Output	Test data	Result	Description
Chk-001	Enter valid shipping information and place a COD order	1. Log in as a user 2. Add products to the cart 3. Navigate to the Checkout page 4. Enter valid name/phone/address 5. Select the COD payment method 6. Click **"Confirm Order"**	Order is placed successfully; an order is created and the order ID is displayed	Valid information	Pass	Decision table/COD
Chk-003	Invalid phone number format	1. Navigate to the Checkout page 2. Enter a phone number containing letters 3. Click **Confirm**	No order is created; a format validation error is displayed	phone='09ab'	Fail	Validation

Chk-004	Shipping address is too short or invalid	1. Navigate to the Checkout page 2. Enter a very short shipping address 3. Click **Confirm**	An invalid address error message is displayed	address='a'	Fail	BVA

(3) Boundary Value Analysis (BVA)

Technique Description
Boundary Value Analysis focuses on testing boundary values, where defects are most likely to occur.

Application to the Checkout Module

- Length of recipient name
- Length of shipping address

Test Case Generation Approach

- Lower boundary values
- Upper boundary values
- Out-of-bound values

Generated Test Case

- Chk-013: Recipient name exceeds the maximum allowed length

ID	Test Case Description	Test Case Procedure	Expected Output	Test data	Result	Description
Chk-013	Recipient name length exceeds the allowed limit	1. Navigate to the Checkout page 2. Enter an excessively long recipient name 3. Click **Confirm**	An error message is displayed or the input is handled according to	name length > max	Fail	BVA

			defined business rules			

(4) Business Rule Testing

Technique Description
Business Rule Testing verifies that the system enforces defined business rules correctly.

Application to the Checkout Module

- Prevent duplicate order creation
- Prevent checkout when inventory is insufficient
- Prevent checkout when the cart is empty

Test Case Generation Approach

- Identify each business rule
- Create test cases that intentionally violate the defined rules

Generated Test Cases

- Chk-005: Double submission (idempotency)
- Chk-006: Inventory changes before order confirmation
- Chk-009: Empty cart checkout

ID	Test Case Description	Test Case Procedure	Expected Output	Test data	Result	Description
Chk-005	Double submission by clicking **"Confirm Order"** twice	1. Navigate to the Checkout page 2. Click **"Confirm"** twice consecutively 3. Check the order list	Only one order is created (idempotent behavior)	double click submit	Fail	Idempotency

Chk-006	Inventory is depleted just before confirmation	1. User A opens checkout with a product having stock = 1 2. User B purchases the product first 3. User A clicks **Confirm** 4. Observe the response	Out-of-stock message is displayed; no order is created	race condition	Pass	Concurrency
Chk-009	Prevent checkout when the cart is empty	1. Open an empty cart 2. Access the Checkout page (if accessible) 3. Click **Confirm**	Checkout is blocked; the system prompts the user to add products	empty cart	Pass	Business rule

(5) Concurrency & Race Condition Testing

Technique Description
This technique evaluates system behavior when multiple actions occur concurrently, particularly in resource contention scenarios.

Application to the Checkout Module

- Multiple users checking out the same product
- Multiple order confirmation requests submitted simultaneously

Test Case Generation Approach

- Simulate concurrent user operations.
- Observe the results of order creation and inventory updates.

Generated Test Case

- Chk-006: Inventory race condition

ID	Test Case Description	Test Case Procedure	Expected Output	Test data	Result	Description
Chk-006	Inventory is depleted just before order confirmation	1. User A opens the checkout page with a product having stock = 1 2. User B purchases the product first 3. User A clicks **Confirm** 4. Observe the system response	An out-of-stock message is displayed; no order is created	race condition	Pass	Concurrency

(6) Integration & Data Integrity Testing (Black-box level)

Technique Description
This technique verifies interactions between modules and data integrity without inspecting internal logic.

Application to the Checkout Module

- Checkout ↔ Cart
- Checkout ↔ Order
- Checkout ↔ Inventory

Test Case Generation Approach

- Modify the data state prior to checkout
- Observe the system behavior and outcomes.

Generated Test Cases

- Chk-007: Product deleted before checkout
- Chk-008: Product price changed before checkout

ID	Test Case Description	Test Case Procedure	Expected Output	Test data	Result	Description
Chk-007	Product in the cart is deleted before checkout	1. User adds a product to the cart 2. Admin deletes the product 3. User proceeds to checkout 4. Observe the system behavior	Checkout fails with a notification prompting the user to update the cart	product deleted	Pass	Integration
Chk-008	Product price is changed before checkout	1. User adds a product to the cart 2. Admin updates the product price 3. User proceeds to checkout 4. Observe the total amount and confirm	The system updates the total amount accordingly	price changed	Fail	Business rule

(7) Error Handling & Reliability Testing

Technique Description

This testing verifies the system's ability to handle errors and maintain reliability under abnormal conditions.

Application to the Checkout Module

- Network disconnection
- Interruption during order confirmation

Test Case Generation Approach

- Simulate environmental failures.
- Verify that the system does not create inconsistent data

Generated Test Case

- Chk-012: Network failure during order confirmation

ID	Test Case Description	Test Case Procedure	Expected Output	Test data	Result	Description
Chk-012	Network failure during order confirmation	1. Navigate to the Checkout page 2. Disable the network connection when clicking **Confirm** 3. Re-enable the network connection 4. Observe the order and cart status	A clear error message is displayed; no inconsistent order data is created	network off	Fail	NFR/Reliability

4.3.2. White-box Testing Methodology

4.3.2.1. Concept of White-box Testing

White-box testing is a testing methodology based on the internal structure of the program. The tester analyzes processing logic, conditional branches, and execution paths in the source code to design test cases.

Unlike black-box testing, which focuses only on inputs and outputs, white-box testing helps to:

- Detect logical defects.
- Ensure that all critical conditional branches are tested.
- Evaluate code coverage.

In the MOW Garden Website project, white-box testing is applied to critical backend business functions to enhance system reliability.

4.3.2.2. Scope of White-box Testing

White-box testing is applied to core backend functions, including:

- Login functionality
- Add to Cart functionality
- Checkout / Place Order functionality
- Inventory / Stock Update functionality

These functions contain multiple conditional branches and directly impact system data.

4.3.2.3 White-box Testing for Checkout / Place Order

A) Logic Description

The order placement process performs the following steps:

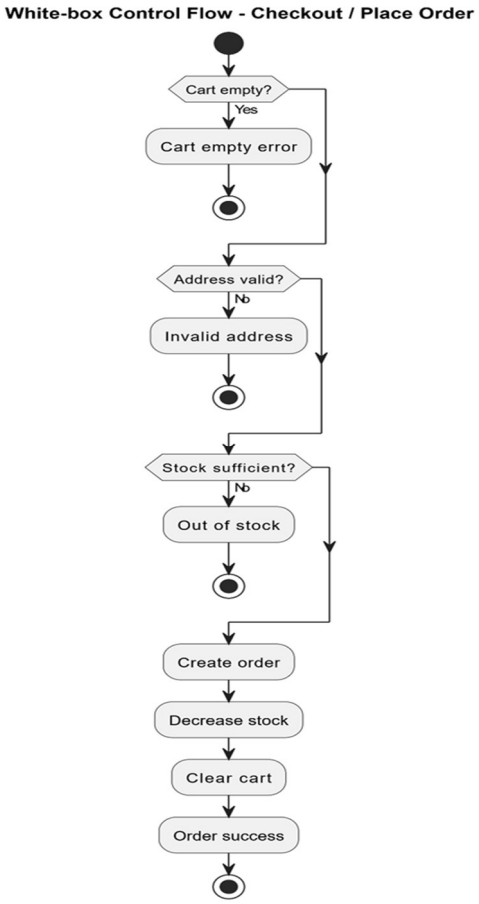
- Validate the shopping cart.
- Validate shipping information.
- Check inventory availability.
- Create the order and update data.

B) Pseudo-code

```
start
if (order not found)
   return error
if (order status = Delivered)
   decrease stock
else if (order status = Canceled)
   restore stock
end
```

C) Independent Paths

Path ID	Description
P1	Empty cart
P2	Invalid address
P3	Insufficient inventory
P4	Successful order placement



Pic 27 Checkout / Place Order

Figure Description

There are **four independent execution paths** for the **Checkout / Place Order** process:

1. **Cart empty?**
   - **Yes** → Cart empty error → Stop
     ✅ *Path P1:* Empty cart
   - **No** → Proceed to the next step
2. **Address valid?**
   - **No** → Invalid address → Stop
     ✅ *Path P2:* Invalid address
   - **Yes** → Proceed to the next step
3. **Stock sufficient?**
   - **No** → Out of stock → Stop
     ✅ *Path P3:* Insufficient inventory
   - **Yes** → Proceed to the next step
4. **Create order → Decrease stock → Clear cart → Order success**
   - Successful execution branch
     ✅ *Path P4:* Successful order placement

D) White-box Testing Evaluation

Through the application of white-box testing:

- All critical conditional branches are thoroughly tested.
- The risk of logical defects in business processing is reduced.
- White-box testing effectively complements black-box testing and automated testing.

Overall, white-box testing helps ensure that the system operates according to its design and enhances the stability of the backend.

4.3.3. Workflow-based Business Testing (End-to-End Testing)

4.3.3.1. Concept of Workflow-based Testing (E2E)

Workflow-based business testing, also known as **End-to-End (E2E) Testing**, is a testing approach that verifies whether the entire system functions correctly when users perform real-world operations from start to finish.

Unlike unit testing and integration testing, which focus on individual components, E2E testing concentrates on the complete business workflow, including:

- User interface (Frontend)
- Backend business logic processing
- End-to-end data flow across system functionalities

4.3.3.2. Objectives of Applying Workflow Testing in the Project

In the MOW Garden project, workflow-based testing is applied with the following objectives:

- Simulating real behaviors of end users.
- Ensuring that system functionalities are correctly linked according to defined business processes.
- Detecting defects that arise when multiple functionalities are used continuously.
- Evaluating the system's readiness before deployment.

4.3.3.3. Scope of Workflow (E2E) Testing

E2E testing is implemented for the core business workflows of the system, including

- User login and authentication.
- Browsing and viewing product listings.
- Adding products to the shopping car.
- Performing the checkout process.
- Displaying order results and order status.

In addition to the main workflow, E2E testing also covers aspects related to **user experience and user interface behavior**.

4.3.3.4. Tested Workflow Groups

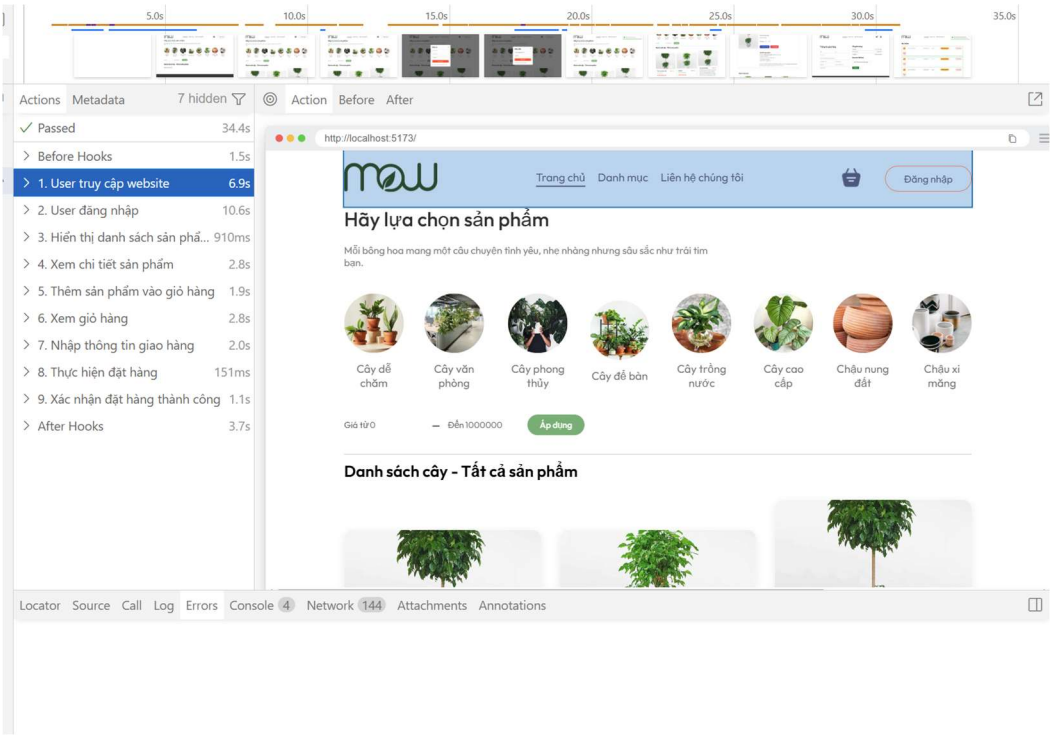
Based on the execution results, the system was tested using **34 E2E test cases**, which are categorized into the following workflow groups:

a) Complete Order Workflow (User Journey)

This workflow tests the primary business flow:

- The user accesses the website.
- The user logs into the system.
- The user browses and views product details.
- The user adds products to the shopping cart.
- The user enters shipping information.
- The user places the order and receives a success notification.

This workflow ensures that the entire purchasing process operates correctly from start to finish.



Pic 28 e2e

The system executes the **E2E User Journey – Full Order Flow** following the sequence:
Login → Browse → View Product → Add to Cart → Checkout → Success.

This represents the **most complete and standard form of E2E testing**, accurately reflecting a real business **workflow**.

b) Form Validation and User Experience Workflow

This workflow focuses on validating user inputs and overall user experience, including:

- Validating data in login and registration forms.
- Displaying appropriate error messages when required data is missing or invalid.
- Preventing users from continuing the workflow when the input data is not valid.

This workflow emphasizes the accuracy and clarity of system feedback to users.

c) User Interface State Handling Workflow

This workflow tests how the user interface responds to different system states, including:

- Loading states while data is being fetched.
- Empty states when no data is available.
- UI responses when users perform continuous or repeated actions.

The objective is to ensure that the user interface accurately reflects the current system state.

d) System Navigation Workflow

This workflow verifies system navigation and access control, including:

- Navigation between frontend pages.
- Navigation within the administrative (Admin) area.
- Role-based access control and permission handling.

This workflow ensures that users are directed to the correct pages with appropriate access rights.

e) Error Notification and System Feedback Workflow

This workflow tests the system's error handling and feedback mechanisms, including:

- Error messages displayed to end users.
- Error notifications displayed in the administrative interface.
- Clarity, accuracy, and contextual relevance of error messages.

f) User Interface Consistency Workflow

This workflow verifies the consistency of the user interface, including:

- Consistency in layout and visual design.
- Responsive behavior across different screen sizes and devices.
- Consistency between the frontend user interface and the administrative interface.

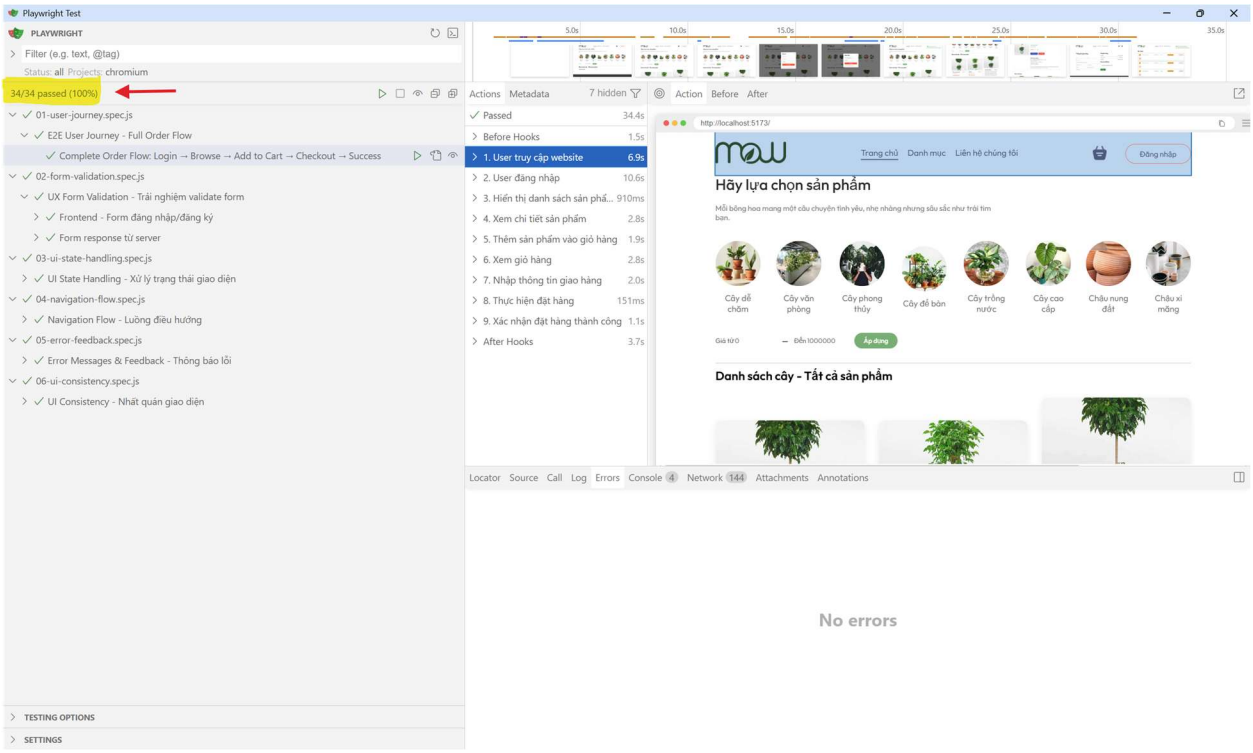
4.3.3.5. Tools and Implementation

End-to-End testing in the project is implemented using **Playwright**.
Test cases are written as source code and executed automatically on the **Chromium** browser.

Playwright enables the following capabilities:

- Simulating real user interactions.
- Executing multiple test cases in parallel.
- Capturing and presenting test results in a clear and visual manner.

4.3.3.6. Test Results



Pic 29 pass e2e

A total of **34 E2E test cases** were executed.
The results show that **100% of the test cases passed**, with no defects detected in the tested business workflows.

This indicates that the system operates stably when users perform real-world actions.

4.3.3.7. Conclusion

Workflow-based business testing (E2E) helps confirm that the MOW Garden system not only functions correctly at the level of individual features but also operates accurately when these features are integrated during real usage.
The application of E2E testing contributes to improving software quality and reducing deployment risks.

4.3.4. Automated Testing and Unit Testing

4.3.4.1. Concept of Automated Testing

Automated Testing is a testing approach that uses testing tools and test scripts to automatically execute test cases, replacing manual testing. Test results are automatically compared between expected outcomes and actual outcomes in order to detect defects quickly and accurately.

In the MOW Garden project, automated testing is implemented at multiple levels:

- **Unit Testing** using Vitest to verify business logic.
- **Integration Testing** to validate APIs, middleware, and database interactions.
- **End-to-End Testing** using Playwright to simulate user behavior.

The application of automated testing helps detect defects early, reduce manual testing effort, and enhance overall system stability.

4.3.4.2 Unit Testing in the MOW Garden Project

1. Objectives of Unit Testing

Unit Testing is performed to verify the correctness of individual functions and business logic in the backend system before these components are integrated.

The main objectives of unit testing in the MOW Garden project include:

- Early detection of logical defects in business processing.
- Ensuring that functions behave correctly for different input scenarios.
- Establishing a stable foundation for integration testing and system testing.

2. Scope of Unit Testing

Unit testing focuses on core backend modules and does not depend on the user interface or connect to real databases or external APIs.

The tested modules include:

- User
- Product
- Cart

- Order
- Validate and error handling

3. Detailed Unit Testing Content

3.1. User Module

Unit test cases are designed to verify:

- Email format validation during registration.
- Password length validation (minimum of 8 characters).
- Handling duplicate email registration.
- Password hashing before persistence.
- Successful login with valid credentials.
- Failed login due to incorrect password or non-existent email.
- Error handling when services encounter failures.

-> These tests ensure that authentication and user-processing logic function correctly.

3.2. Product Module

Unit testing focuses on:

- Adding products with valid data.
- Validating required fields when creating a product.
- Handling invalid data before persistenc.

The objective is to ensure that product-processing logic is correct before interacting with the database.

3.3. Cart Module

Unit test cases verify:

- Adding new products to the cart.
- Increasing product quantity when the product already exists in the cart.
- Decreasing product quantity in the cart.

- Preventing quantities from being reduced below zero.

These tests ensure that cart operations are handled correctly.

3.4. Order Module

Unit testing is applied to:

- Rejecting order placement when the cart is empty.
- Preventing status updates when an order has already been completed (Delivered).
- Automatically updating payment status when an order is completed.

These tests ensure that business constraints in order processing are enforced correctly.

3.5. Validation and Error Handling

In addition to core modules, unit testing also verifies:

- Error handling when services throw exceptions.
- Validation of required fields in critical APIs.
- Ensuring that the system does not crash when errors occur.

4. Characteristics of Unit Testing in the Project

- Test cases are written as source code and executed automatically.

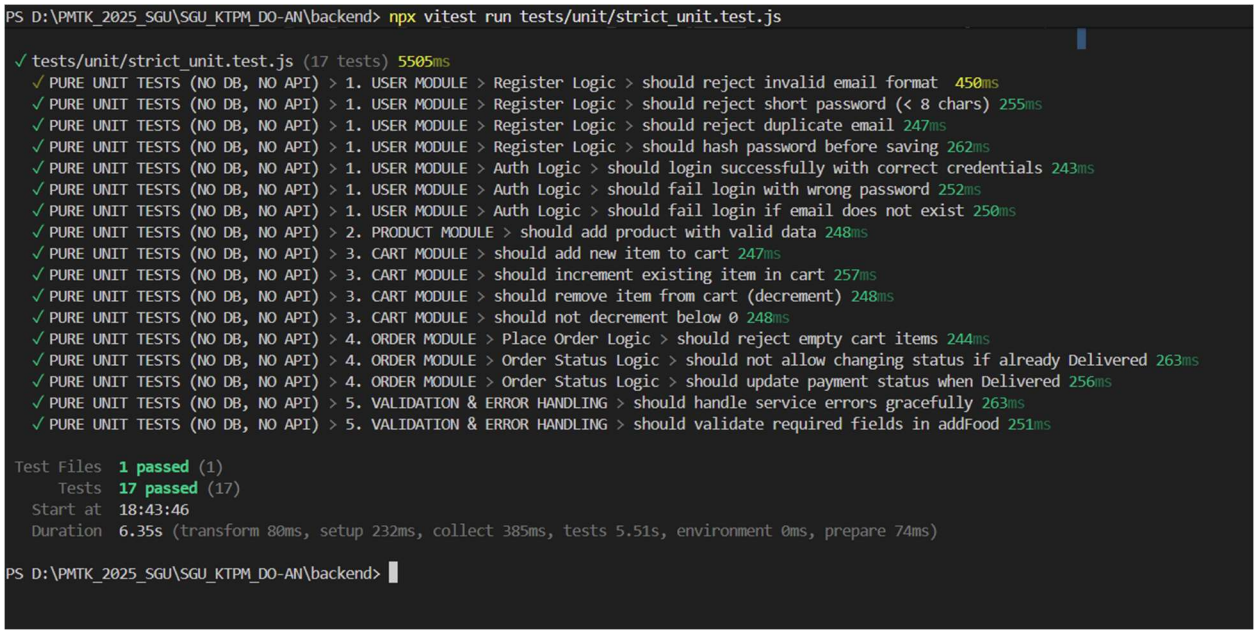
- No connection to a real database.

- No dependency on external APIs.

- Dependent components are mocked.

- Each test case validates a specific logic scenario.

5. Conclusion



Pic 30 .unit tess

Through the application of **unit testing**, the MOW Garden project ensures that critical backend business logic functions correctly at the lowest level. This helps reduce the risk of defects during system integration and improves overall software quality.
A total of **17 unit test cases** were executed, with **17/17 test cases passing**

4.4. Application of GenAI in Test Design

4.4.1. Objectives of Applying GenAI in Test Design

In the MOW Garden project, **GenAI** is applied to support the **Test Design phase**, specifically to automatically generate test scenarios for backend controllers based on existing source code.

The application of GenAI in Test Design aims to:

- Assist testers in quickly building an initial list of test scenarios.
- Suggest potential test scenarios based on the actual processing logic of the controllers.
- Reduce manual test case design time, especially for backend modules with multiple conditional branches.
- Increase test coverage at the design stage.

Within the scope of this project, GenAI is used **only for test scenario generation** and does not participate in test execution or result evaluation.

4.4.2. Technologies and Tools Used

The GenAI solution is integrated directly into the backend of the system using the following technologies:

- **Large Language Model (LLM):** Google Gemini API
- **Implementation Language:** JavaScript (Node.js)
- **Integration Location:** backend/ai-tools directory
- **Usage Method:** Internal project CLI command

GenAI is utilized as a Test Design support tool, enabling testers to proactively generate test scenarios from specific controllers.

4.4.3. GenAI-based Test Scenario Generation Process

The GenAI-assisted Test Design process is carried out through the following steps:

Step 1: Select the Target Controller

The tester identifies the backend controller for which test scenarios need to be generated.
Example in this project:

userController

This controller handles functionalities related to user registration, login, and authentication.

Step 2: Execute the Test Scenario Generation Command

The tester executes the CLI command provided by the system:

npm run ai:generate -- --controller userController

When this command is executed, the system will:

- Read and analyze the source code of userController
- Extract API endpoints, processing logic, and validation condition

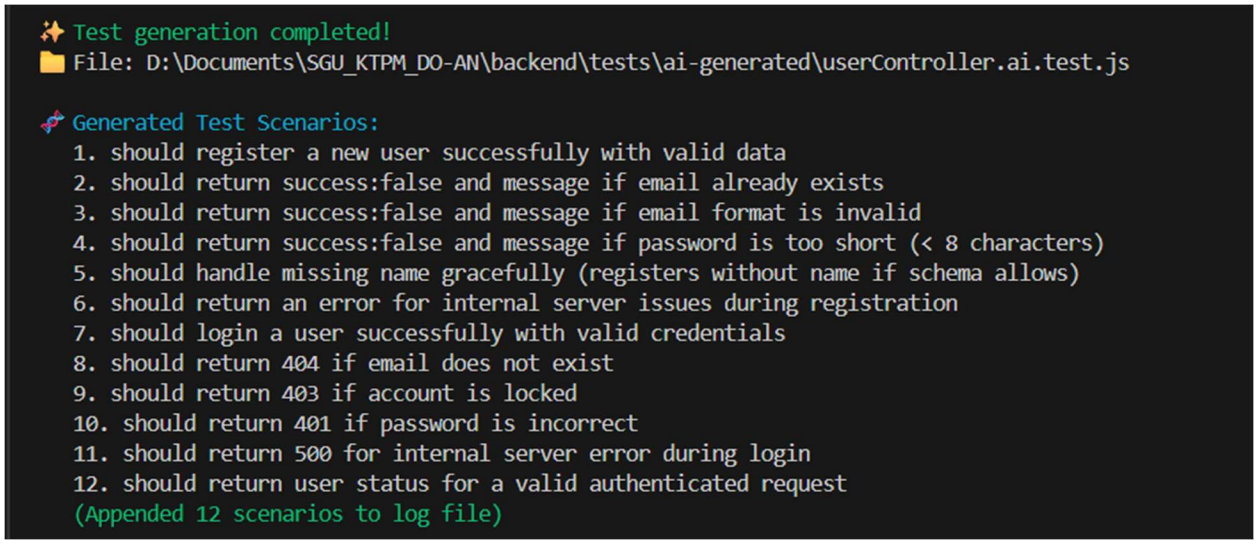
- Send the analyzed content to the Gemini API to generate test scenarios

Step 3: Generate Test Scenarios

Based on the controller source code, GenAI generates a list of test scenarios in descriptive form, including:

- Successful processing scenarios (happy paths)
- Invalid input data scenarios
- Common business logic error scenarios
- System error scenarios (e.g., internal server errors)

The generated test scenarios are **not executable test scripts** but serve as outputs of the **Test Design phase**.



Pic 31 genAI

Step 4: Storing Generated Test Scenarios

All test scenarios generated by GenAI are recorded and stored in a separate file to support review and refinement of the Test Design.

In the project, these generated test scenarios are stored in the following directory:

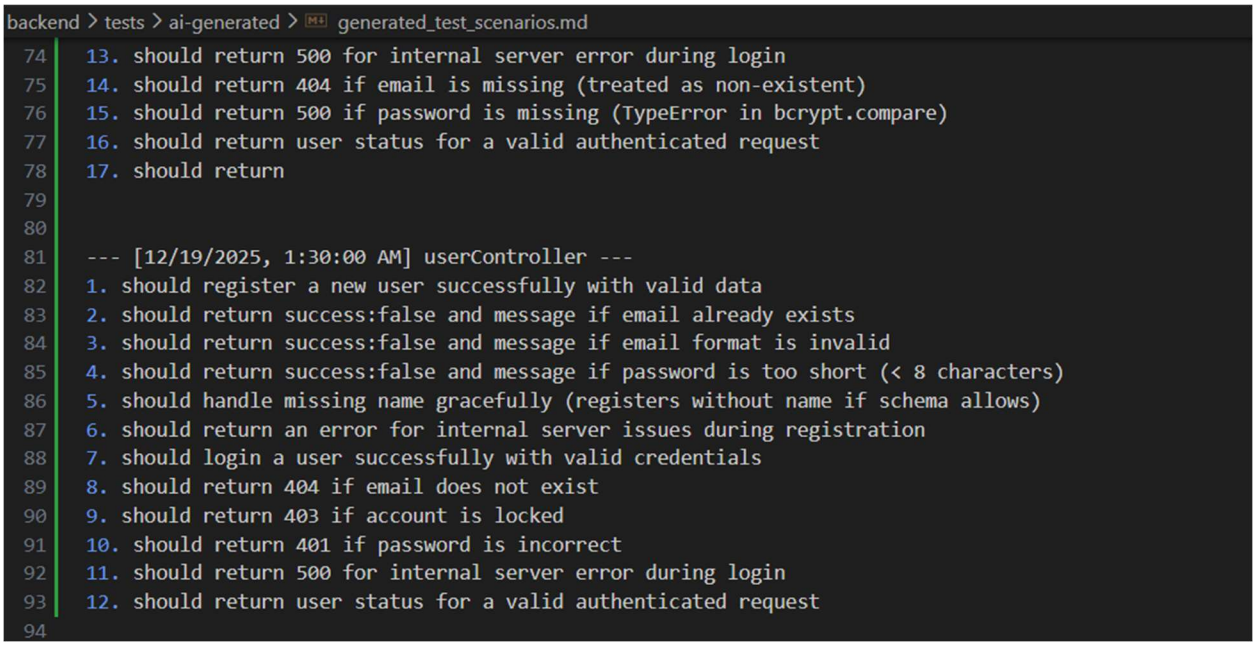
backend/tests/ai-generated/

For example, when generating test scenarios for the userController, the system updates the file:

generated_test_scenarios.md

This file contains:

- A list of generated test scenarios for each controller
- The timestamp indicating when the test scenarios were generated
- Detailed descriptions of the testing scenarios



Pic 32 test scenario

4.4.4. Role of Test Scenario Files in Test Design

The test scenario files generated by GenAI serve the following purposes:

- They represent the output of the GenAI-assisted Test Design process
- They act as reference materials for testers when constructing formal test cases
- They provide a basis for selecting and standardizing test cases in Test Design documents (Excel/Word)

The test scenarios in these files:

- Are **not used directly for test execution**
- Do **not replace tester-designed test case**
- Must be **reviewed, refined, and selectively approved** by testers before use

4.4.5. Scope of Application and Limitations

Within the scope of **Chapter 4 – Test Design**, GenAI is applied with clearly defined boundaries:

Scope of Application

- Generating test scenarios for backend controllers
- Supporting the test design phase
- Suggesting testing scenarios based on source code analysis

Limitations

- Does not execute tests
- Does not evaluate pass/fail results
- Does not replace Unit Testing, Integration Testing, or Automated Testing
- Does not automatically apply generated test scenarios to the system.

4.4.6. Conclusion

The application of GenAI (Gemini API) in Test Design enhances the efficiency of test design in the MOW Garden project, particularly in generating test scenarios for backend controllers.

GenAI is used as an intelligent support tool that helps testers save time and increase test coverage, while ensuring that testers remain fully responsible for the final quality of the Test Design.

CHAPTER: Testing Result

5.1. Illustration of Test Execution

5.1.1. Test Execution Scope

The test execution was conducted on the main modules of the system, including:

- Functional Testing
- Crowd Testing
- Usability Testing
- Interface Testing
- Security Testing
- Performance Testing
- Compatibility Testing
- Database Testing

The test cases were executed based on:

- **Test Design document (Word)**
- **Detailed Test Cases (Excel)**

5.2.2. Tools and Test Execution Environment

Testing Tools

The tools used during the test execution process include:

- **Postman:** Manual API testing
- **Vitest / Jest + Supertest:** Automated API testing
- **JMeter:** Performance testing
- **OWASP ZAP:** Basic security testing
- **GenAI (Gemini API):** Support for generating test scenarios (not used for test execution)

Test Execution Environment

- **Backend:** Node.js + Express
- **Database:** MongoDB
- **Test Environment:** Local environment

5.2.3. Illustration of Test Execution

5.2.3.1. API Test Case Execution Using Postman

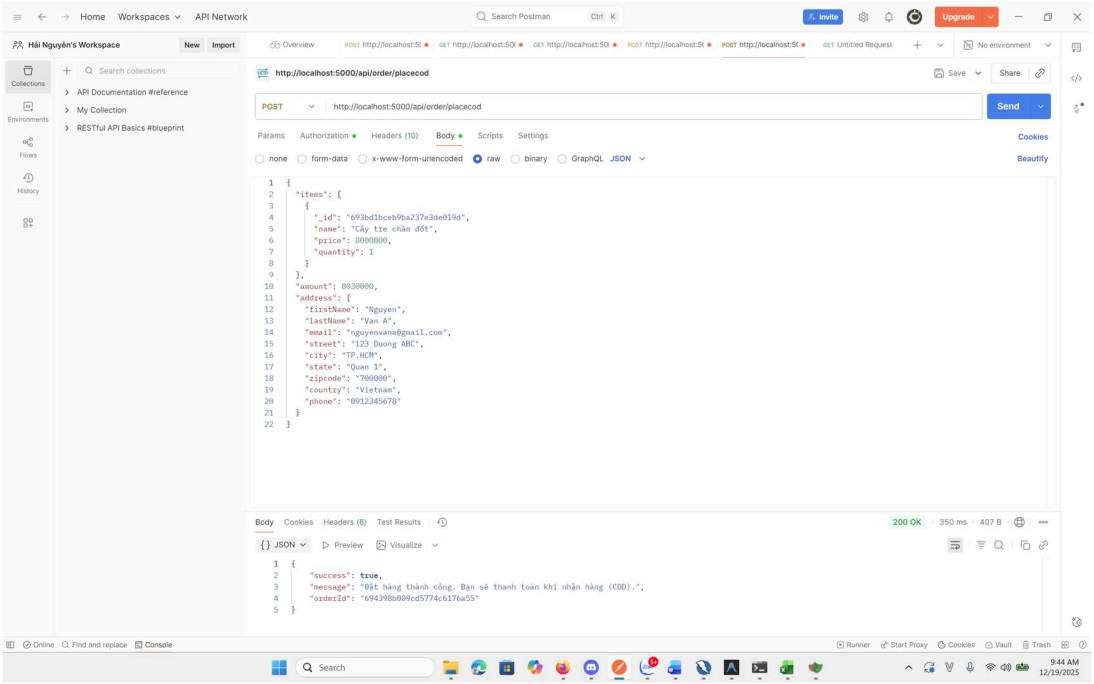
API: POST http://localhost:5000/api/order/placecod

Test Objective

This test case is executed to verify the **Cash on Delivery (COD)** order placement functionality of the **MOW Garden** system through the backend API.
The primary objective is to validate that the API correctly handles the order business logic, including user authentication, input data validation, inventory processing, and order creation.

API Information

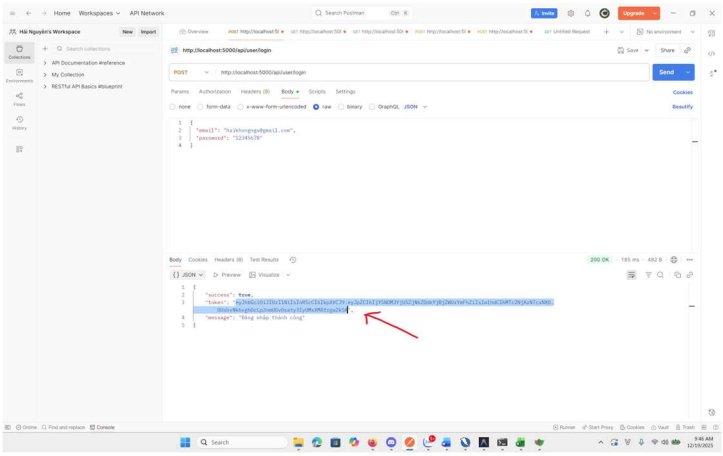
- **Endpoint:** POST /api/order/placecod
- **URL:** http://localhost:5000/api/order/placecod
- **Method:** POST
- **Authentication Required:** Yes (JWT Token)
- **Execution Tool:** Postman



Input Data (Request)

Header

Token Source



- Authorization: Bearer <eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjY5NDM3YjU5ZjNkZDdkYjBjZWUxYmFhZiIsImlhdCI6MTc2NjAzNTcxNX0.UUshvNk6vghOcLp3nmUGvOsaty3IyUMxXM8frga2k5A
- >
- Content-Type: application/json

Body(json)


```
{
 "items": [
  {
   "_id": "693bd1bceb9ba237e3de019d",
   "name": "Cây tre chăm đốt",
   "price": 8000000,
   "quantity": 1
  }
 ],
 "amount": 8030000,
 "address": {
  "firstName": "Nguyen",
  "lastName": "Van A",
  "email": "nguyenvana@gmail.com",
  "street": "123 Duong ABC",
  "city": "TP.HCM",
  "state": "Quan 1",
  "zipcode": "700000",
  "country": "Vietnam",
  "phone": "0912345678"
 }
}
```


Postman Execution Steps

1. Open Postman and create a new request with the **POST** method.
2. Enter the URL: http://localhost:5000/api/order/placecod.
3. Add the **Authorization** header with a valid **JWT Token**.
4. Select **Body → raw → JSON** and enter the request data.
5. Click **Send** to submit the request to the backend.

Expected Result

- The API returns HTTP status **200 OK** or **201 Created**.
- The response body contains information about the newly created order.
- The order is saved in the database with the initial status **Processing**.
- The product inventory quantity is reduced accordingly.
- The user's shopping cart is cleared after a successful order placement.

Actual Result

After executing the test case in Postman, the API returned the expected results:

- The order was created successfully.
- No errors occurred during the processing.
- Data was updated accurately in the system.

Test Case Status: PASS

Conclusion

The test case **POST /api/order/placecod** demonstrates that the **Cash on Delivery (COD)** order placement functionality of the **MOW Garden** system operates correctly according to business requirements.
The API accurately processes input data, properly authenticates users, and updates the system state as expected, thereby ensuring the reliability of the **Order** module.

5.2.3.2. Automated Testing Using Vitest (npm test)

In this project, when executing automated tests using **Vitest (npm test)**, the team performed two main types of testing: **Unit Testing** and **Integration Testing**.

Testing Types Overview

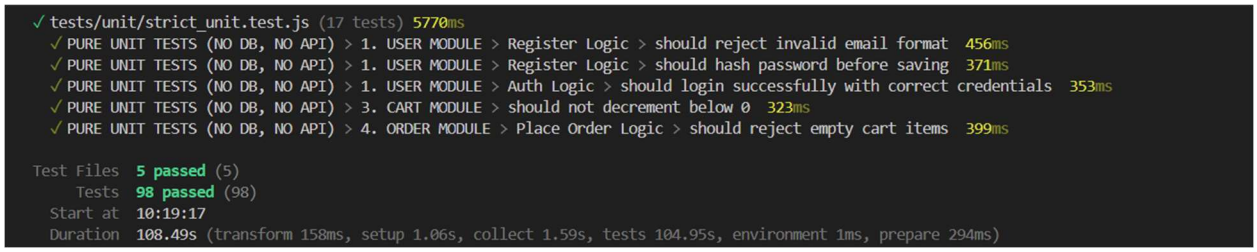
- **Unit Testing:**
  Includes **17 test cases**, which are used to verify the business logic of

individual modules (User, Product, Cart, Order).
These tests do not connect to a real database and do not call external APIs.

- **Integration Testing:**
  Verifies the interaction and coordination between backend modules through APIs.
  These tests use a test database and real test data to validate end-to-end business workflows.

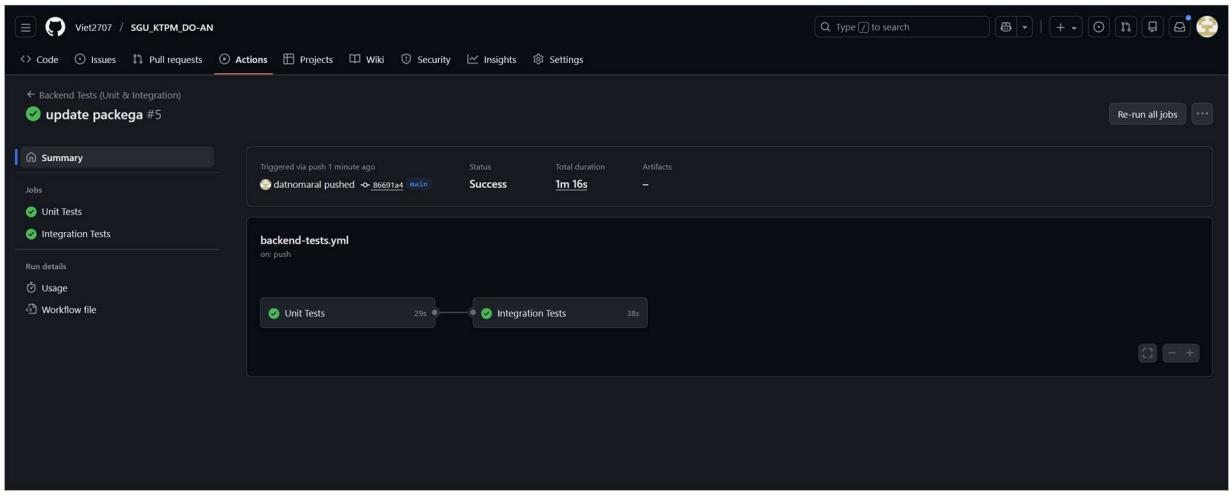
Test Execution

When running the npm test command in **PowerShell**, the automated test suite is executed, including both Unit Tests and Integration Tests.



Pic 33 npm test

17 test case unit test and 81 intergation test



Automated Test Execution via CI/CD (GitHub Actions)

The figure above illustrates the automated testing process in the CI/CD pipeline of the **MOW Garden** project using **GitHub Actions**.
Whenever source code changes are pushed to the **main** branch, the CI/CD system automatically triggers the **backend-tests.yml** workflow.

This pipeline includes two main testing stages:

- **Unit Tests:**
  Execute unit test cases to verify the business logic of each backend module independently.
  This stage helps detect logic errors early before integrating components.

- **Integration Tests:**
  After the Unit Tests pass successfully, the system continues with integration testing to validate the interaction between modules through APIs, ensuring that business workflows operate correctly when components are connected.

Observed Results:

- Both **Unit Tests** and **Integration Tests** completed successfully (**Success**).
- The total pipeline execution time was **1 minute and 16 seconds**.
- These results indicate that the backend system is stable and ready for subsequent deployment stages.

Integrating automated testing into the CI/CD pipeline helps ensure software quality, reduces the risk of defects during code updates, and improves overall system reliability.

5.2.3.3. Performance Testing Using JMeter

a. Performance Test Scenarios

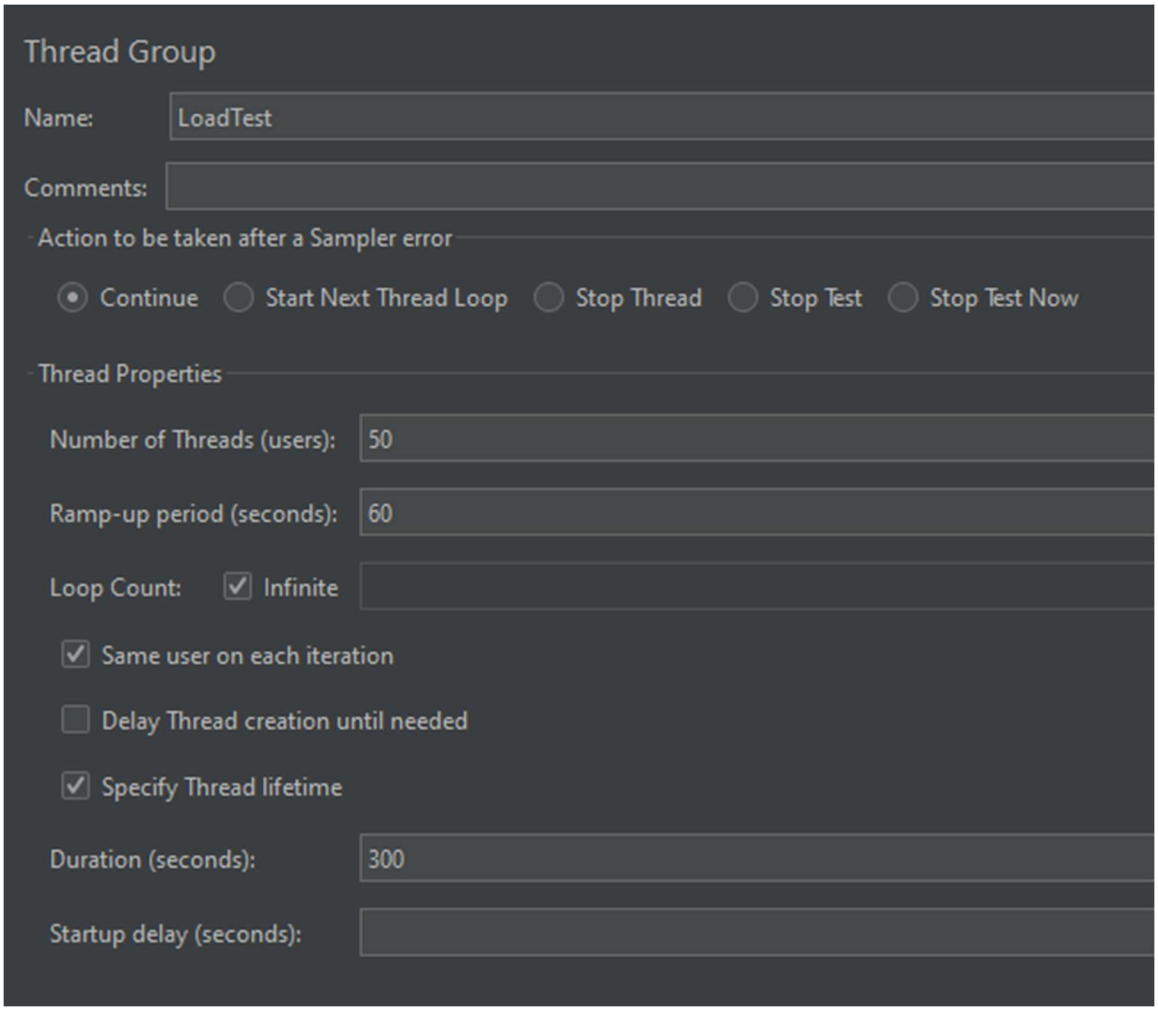
Within the scope of this project, two main performance testing scenarios were conducted.

Scenario 1: Homepage Load Test (PERF-01)

Objective:
Evaluate the system's ability to handle concurrent access to the homepage.

Test Configuration:

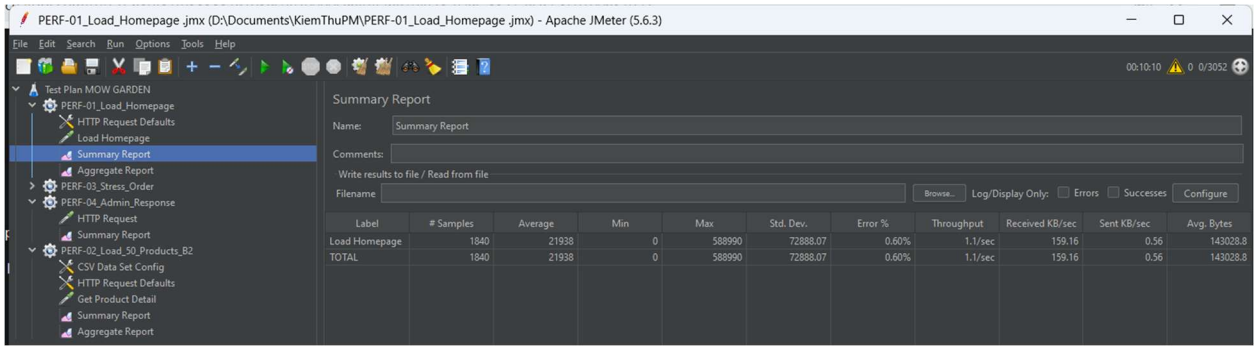
- **Number of concurrent users:** 50 users
- **Test duration:** 5 minutes
- **Tool:** JMeter Thread Group
- **Request type:** HTTP GET request to access the homepage



Pic 34 Scenario 1 Jmeter

Measured Results:

- **Average Response Time:** ~21,938 ms
- **Maximum Response Time:** ~588,990 ms
- **Throughput:** ~1.1 requests/second
- **Error Rate:** ~0.60%



Pic 35 jMeter Result

Evaluation Result:

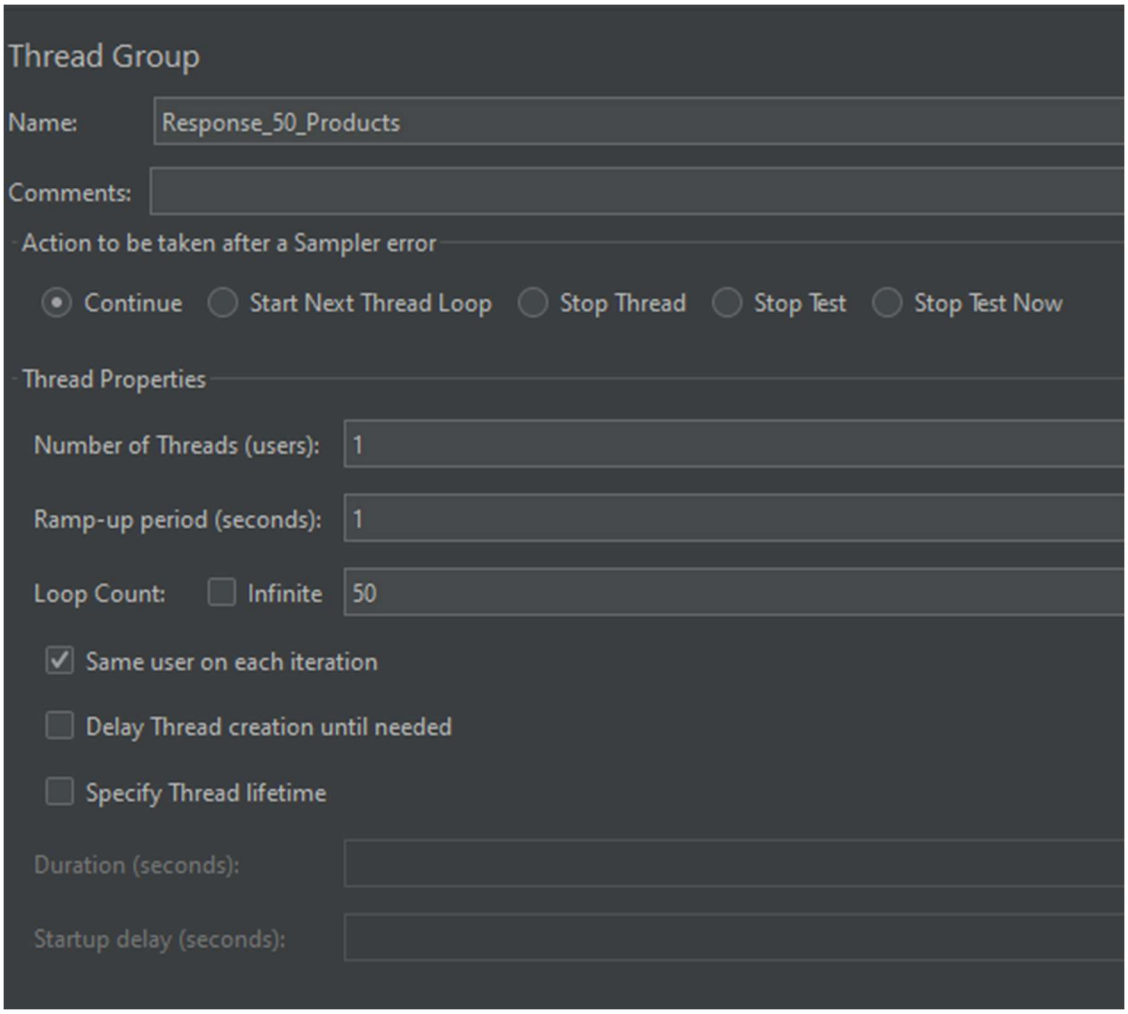
Fail – The average response time exceeds the expected threshold (**< 3000 ms**).

Scenario 2: Product Detail Load Test (PERF-02)

Objective:
Evaluate system performance when users repeatedly access product detail pages.

Test Configuration:

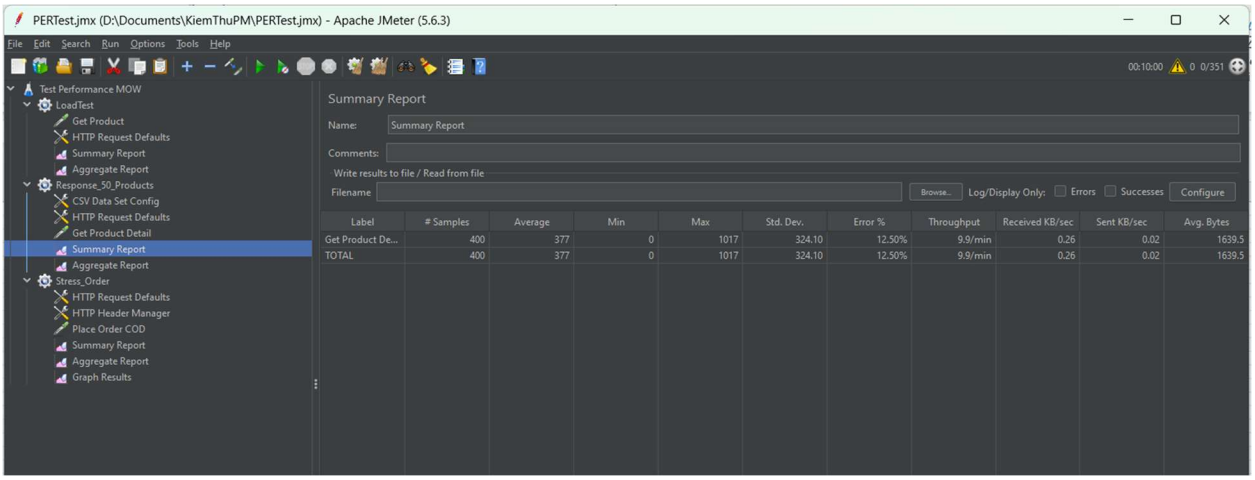
- **Number of requests:** 50 sequential requests
- **Number of users:** 1 user
- **Request type:** HTTP GET (product detail)
- **Input data:** List of productId values loaded from a CSV file



Pic 36 Scenario 2 jMeter

Measured Results:

- **Average Response Time:** ~377 ms
- **Maximum Response Time:** ~1017 ms
- **Throughput:** ~9.9 requests/minute
- **Error Rate:** ~12.5%



Pic 37 Scenario 2 jMeter Result

Evaluation Result:

Pass – The average response time meets the required threshold (**< 3000 ms**).
The system performs well under low load conditions and sequential access.

C. Observations and Evaluation

Based on the performance testing results, the following observations can be made:

- The system operates stably under low load or sequential access conditions.
- Performance degrades significantly as the number of concurrent users increases.
- The homepage is a critical area that requires further optimization to improve response time.
- The error rate remains within an acceptable range but should continue to be monitored.

5.3. Test Report

After completing the test execution, the summarized results are as follows:

No.	Test Type	Test Objective	Total Test Cases	Pass	Fail	Untested
1	Functional Test	Verify core business functionalities of the system	80	64	15	1
2	API Test	Verify backend APIs (request/response, status codes)	39	39	0	0
3	UI Test	Verify user interface and user interactions	31	31	0	0
4	Acceptance Test	Confirm the system meets user requirements	5	5	0	0
5	Performance Test	Evaluate system performance and load handling	2	1	1	0
6	Security Testing	Verify basic security issues	81	81	0	0
7	Usability Test	Evaluate system usability	6	6	0	0
8	Compatibility Test	Verify browser and device compatibility	6	5	1	0
9	Crowd Test	Collect feedback from multiple users	5	0	0	5
10	Unit Test	Verify correctness of functions, modules, or controllers at unit level	17	17	0	0
11	Integration Test	Verify interaction and data exchange between modules	8	8	0	0
12	End-to-End (E2E) Test	Verify complete business workflows from start to end	12	12	0	0

The test results indicate that the system satisfies most of the defined functional and non-functional requirements.

5.4. Defect Report

During the test execution process, defects were recorded following the steps below:

1. Detect defects during test case execution
2. Log defects into the **Defect List**
3. Classify defects based on **Severity**
4. Track defect resolution status

Defect ID	Defect Description & Steps to Reproduce	Actual Result	Expected Result	Priority	Severity	Test Case ID
#3	**Price filter allows min price greater than max price** 1. Access the product list page. 2. Enter a minimum price greater than the maximum price. 3. Click the filter button.	The system still performs the filtering without any warning.	The system should display an error message: *"Minimum price must not be greater than maximum price."*	Medium	Major	Cat-007
#4	**Admin allows adding a**	The system allows the	The system should reject the	High	Critical	Cat-014

Defect ID	Defect Description & Steps to Reproduce	Actual Result	Expected Result	Priority	Severity	Test Case ID
	product with a negative price 1. Log in as Admin. 2. Navigate to the Add Product page. 3. Enter price = -10000. 4. Click Save.	product to be saved.	input and display an error message: *"Invalid price."*			
#5	**Uploading non-image files when adding a product** 1. Log in as Admin. 2. Navigate to the Add Product page. 3. Upload a .txt or .exe file. 4. Click Save.	The system allows the file to be uploaded.	The system should only allow image files (.jpg, .jpeg, .png) and display a clear error message.	High	Major	Cat-015

5.5. Test Summary Report

5.5.1. Requirement Fulfillment Evaluation

Based on the testing results, the system:

- Successfully meets the core functional requirements.

- Satisfies non-functional requirements at an acceptable level.
- Can be deployed for trial use in a real environment with some additional improvements.

5.5.2. Test Design Effectiveness Evaluation

The Test Design, developed in **Chapter 4**, demonstrates the following strengths:

- Fully covers critical business workflows.
- Helps detect business logic issues and exception handling problems at an early stage.
- Provides strong support for both manual and automated testing.

In particular, the application of **GenAI in Test Design** contributes to:

- Accelerating test scenario creation.
- Expanding test coverage.
- Assisting testers in identifying potential and edge-case scenarios.

5.5.3. Limitations and Future Improvements

Limitations:

- Some test cases could not be executed due to environment dependencies.
- Security testing was not performed at an advanced level.
- Automation testing does not yet cover all test cases.

Future Improvements:

- Expand automation testing coverage.
- Enhance security testing.
- Optimize the process of applying GenAI in software testing.

5.6. Conclusion

Chapter 5 has fully presented the testing results of the **MOW Garden** system. The results show that the system meets most of the defined requirements, the Test Design is well-structured, and the testing process delivers practical and effective outcomes.