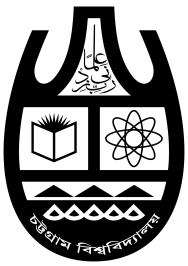
University of Chittagong
Department of Computer Science and Engineering



Warehouse Management System

Course : Software Engineering

Course Code : CSE - 516

Submitted to,

Dr. Mohammad Osiur Rahman
Professor
Department of Computer Science and Engineering
University of Chittagong

Submitted by,

Group – 19

Sl. No.	Name	ID
1	Ahsanul Hoque	22701048
2	Wazidul Alam	22701045
3	Abroy Shoban Chowdhury	22701053

Date of Submission: August 20, 2025

Declaration

We, the undersigned team members, hereby declare that the project work entitled "**Warehouse Management System**" is our original work and has not been submitted elsewhere for any other degree or diploma.

Name	**Signature**
Ahsanul Hoque	
Wazidul Alam	
Abroy Shoban Chowdhury	

Acknowledgment

We would like to express our sincere gratitude to all those who have contributed to the successful completion of this project.

First and foremost, we are deeply grateful to our supervisor, **Dr. Mohammad Osiur Rahman**, for their invaluable guidance, encouragement, and continuous support throughout the project. Their insights and expertise greatly enhanced the quality of our work.

We would also like to thank the faculty members of the **Computer Science and engineering Department**, whose knowledge and teachings provided the foundation for this project.

Finally, we extend our heartfelt appreciation to our friends and peers who offered assistance, feedback, and motivation during the development of the Warehouse Management System. Their support was instrumental in overcoming challenges and achieving the project goals.

This project would not have been possible without the collective support and encouragement of all these individuals.

Abstract

Efficient stock management is vital for companies to maintain accurate inventory levels, minimize wastage, and meet customer demands. This project addresses the challenges faced by businesses in manually tracking inventory, which often leads to errors, stock discrepancies, and delays in fulfilling orders. Current solutions, typically relying on outdated software or manual methods, lack real-time data updates and scalability, resulting in inefficiencies and poor user experiences. To bridge this gap, this project develops an integrated web-based inventory management system using modern technologies, including MySQL for backend storage, Node.js for backend logic, and a user-friendly front-end interface. The system will feature real-time stock tracking, automated order processing, inventory alerts, and business analytic suggestions powered by generative AI. Additionally, an AI chatbot will enhance user interaction, while detailed reporting capabilities will assist businesses in making informed decisions. The system's effectiveness will be evaluated through tests on database operations, usability, and feedback from stakeholders, with success metrics including stock tracking accuracy, system uptime, and user satisfaction. Although the initial complexity of the implementation and reliance on internet connectivity present challenges, the system will significantly enhance operational efficiency. Future work will include integrating AI-based predictive analytics for demand forecasting.

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
MVC	Model-View-Controller
WMS	Warehouse Management System
FR	Functional Requirement
NFR	Non-Functional Requirement
IoT	Internet of Things
UAT	User Acceptance Testing
ER	Entity Relationship
NF	Normal Form
CRUD	Create, Read, Update, Delete
HTML	Hypertext Markup Language
AI	Artificial Intelligence
SQL	Structured Query Language
EJS	Embedded JavaScript
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
GUI	Graphical User Interface
RESTful	Representational State Transfer
UI	User Interface
CI/CD	Continuous Integration and Continuous Delivery
RFID	Radio Frequency Identification

Chapter 1

Introduction

1.1 Background

The significance of this project lies in its ability to automate and streamline essential warehouse tasks. By replacing manual processes with digital solutions, it improves operational efficiency, reduces errors, and ensures faster and more accurate order fulfillment. Furthermore, the system supports both distributors and customers, enhancing communication and reducing operational overhead.

In today's competitive business environment, warehouses play a critical role in ensuring efficient supply chain operations. Traditional warehouse management systems, often based on manual record-keeping or standalone desktop applications, struggle to cope with the growing complexity of modern businesses. Some of the key challenges faced by such systems include:

- Inventory mismanagement and inaccurate stock levels
- Delayed order processing and fulfillment
- Lack of real-time data visibility for decision-making

With the rising demand for speed, accuracy, and cost-efficiency, organizations are increasingly turning to web-based Warehouse Management Systems (WMS). These systems leverage modern web technologies, real-time analytics, and secure databases to optimize warehouse operations. By addressing the limitations of traditional systems, web-based WMS solutions

not only improve operational efficiency but also significantly enhance customer satisfaction.

1.2 Problem Statement

The problem this project addresses is the inefficiency and error-prone nature of managing inventory and customer data manually in a warehouse setting. Current systems either lack the necessary automation or are too complex for small to medium-sized businesses to implement and maintain. This creates difficulties in ensuring accurate records, efficient workflows, and customer satisfaction.

The key entities involved in this system are:

- **Products:** Items in the warehouse that must be tracked for stock levels, pricing, and order fulfillment.
- **Customers:** Individuals or organizations that purchase products from the warehouse.
- **Orders:** Requests made by customers to purchase products, which require approval from distributors and processing of inventory.
- **Distributors:** Warehouse managers or employees responsible for managing inventory, approving orders, and ensuring the proper delivery of products.

The relationships between these entities involve managing customer orders, tracking product availability, and adjusting stock levels upon purchase. However, manual warehouse management processes often result in:

- Inaccurate stock records and inventory mismanagement
- Misplaced inventory and lack of real-time tracking
- Delayed billing and order fulfillment
- Poor decision-making due to lack of data visibility

These challenges directly affect business performance by increasing operational costs, reducing scalability, and leading to customer dissatisfaction. Furthermore, existing solutions often lack proper integration between inventory management, sales analysis, and billing, which further limits their effectiveness.

Therefore, there is a pressing need for a centralized, real-time, and scalable Warehouse Management System (WMS) that can automate and streamline warehouse operations, minimize errors, and improve overall efficiency while remaining suitable for small to medium-sized businesses.

1.3 Objectives

The objectives of this project are as follows:

- To design and implement a **web-based Warehouse Management System** using Node.js, Express.js, and MySQL.
- To automate critical operations such as **inventory management, billing, and stock analysis**.
- To provide a **real-time dashboard** for monitoring business performance and key warehouse metrics.
- To ensure **secure authentication and authorization** for different types of users.
- To develop **data-driven reporting and analytics features** to support managerial decision-making.
- To integrate an **AI chatbot module** for personalized assistance, insights, and operational recommendations.

1.4 Scope and Limitations

Scope:

- The system includes **inventory management, billing, order tracking, and sales/stock analysis**.
- It features **role-based user authentication**, data visualization, and printable invoice generation.
- The project emphasizes **scalability**, ensuring that it can support growth in both transaction volume and user base.

Limitations:

- The system currently requires **internet connectivity** to operate, as it is hosted online.
- Hardware integrations (e.g., barcode scanners, IoT sensors) are not yet implemented.
- Advanced AI features such as **predictive forecasting** are planned for future releases but not part of the current scope.

1.5 Significance

This project addresses the increasing demand for **digitized and automated warehouse operations**. By providing real-time insights, accurate stock tracking, and streamlined billing, the WMS empowers businesses to make better decisions, reduce operational inefficiencies, and improve customer satisfaction. Furthermore, its modular design ensures long-term maintainability and adaptability, allowing businesses to expand the system with emerging technologies such as **AI-driven analytics** and **IoT-enabled tracking**. Ultimately, the project contributes to both **academic learning in software engineering** and **practical applications in business management**.

Chapter 2

Requirement Analysis

2.1 Requirement Gathering

The requirements for the Warehouse Management System (WMS) were gathered through a combination of stakeholder interviews, document analysis, and observation of existing warehouse practices. The goal was to understand the pain points of manual systems and to identify the essential features for an automated solution.

2.1.1 Interviews

Interviews were conducted with warehouse staff, managers, and administrative personnel to gather insights into their daily workflows and challenges. Key focus areas included:

- How inventory is currently tracked and updated.
- The process of generating bills and invoices.
- Pain points in order management and stock analysis.
- Security concerns regarding access control and data integrity.
- Desired features in a modern WMS, such as dashboards and analytics.

2.1.2 Outcome of Interviews

The outcome of the interviews revealed the following requirements:

- An easy-to-use system for managing products, categories, brands, and stock levels.
- A secure authentication system with different user roles.
- Real-time visibility of inventory status and low-stock alerts.
- Automated billing with printable invoices.
- Analytical tools for understanding sales patterns and stock movement.
- Future interest in integrating AI features and hardware devices such as barcode scanners.

2.2 Functional Requirements

The core functional requirements (FRs) of the WMS are as follows:

- **User Authentication and Authorization:** Secure login, registration, and role-based access control using Passport.js.
- **Dashboard and Real-time Analytics:** A dashboard showing total sales, orders, stocks, and stock value with visual graphs.
- **Inventory Management:** Add, view, update, and delete product details (name, category, brand, size, stock, price).
- **Master Data Management:** Manage categories, brands, and sizes efficiently.
- **Billing and Order Management:** Generate invoices, auto-fill product details, and provide printable bills.
- **Sales and Stock Analysis:** Detailed reporting with filters (monthly, yearly, category-wise trends).
- **AI Chatbot (Advanced Feature):** Assist users with insights, recommendations, and trend exploration.

2.3 Non-Functional Requirements

The non-functional requirements (NFRs) that ensure system quality include:

- **Performance:** Real-time updates with minimal latency.
- **Scalability:** Ability to handle increased products, transactions, and users.
- **Security:** Password hashing with bcrypt, secure sessions, and protection from common attacks.
- **Usability:** A clean, responsive, and intuitive interface accessible from multiple devices.
- **Reliability:** Accurate data and consistent system availability.
- **Maintainability:** Modular, well-documented code to support future enhancements.
- **Availability:** Hosted online, with plans for an offline-ready version in the future.

2.4 Stakeholder Analysis

The key stakeholders of the project include:

- **Warehouse Staff:** End-users who will manage inventory and generate invoices.
- **Warehouse Manager:** Responsible for monitoring operations and analyzing sales reports.
- **System Administrators:** Handle system maintenance, database integrity, and user management.
- **Customers:** Indirect stakeholders who benefit from faster deliveries and improved service quality.
- **Developers and Project Team:** Responsible for designing, implementing, testing, and deploying the system.

2.5 Feasibility Study

The feasibility of the project was evaluated across four dimensions:

- **Technical Feasibility:** The system is feasible using Node.js, Express.js, MySQL, and EJS, supported by widely available libraries.

- **Economic Feasibility:** The project is cost-effective, leveraging open-source technologies and requiring minimal infrastructure.
- **Operational Feasibility:** The system aligns with existing workflows and addresses the pain points of manual processes.
- **Schedule Feasibility:** The modular development approach ensures that the system can be developed and deployed within a reasonable timeframe.

Chapter 3

Team Structure and Project Workflow

3.1 Team Structure

The Warehouse Management System was developed by a three-member team, where each member was assigned a specialized role to ensure division of responsibilities and efficient collaboration. The team structure is as follows:

Team Members	**Role**
Ahsanul Hoque	Full-Stack Developer (Backend and Frontend)
Wazidul Alam	Database Developer and QA Engineer
Abroy Shoban Chowdhury	Deployment and Maintenance Engineer

Table 3.1: Project Team

The team collaborated using GitHub as the central repository. GitHub Projects and Issues were used for task tracking, progress monitoring, and version control. Regular meetings were held to synchronize efforts and align with agile principles.

3.2 Project Workflow

The project followed a structured ten-week workflow to ensure timely delivery and systematic development. Each week focused on specific deliverables, as outlined below.

Week 1: Project Initialization and Setup

Tasks:

- Establish the overall project structure and repository.
- Provide instructions on documentation format, coding guidelines, and workflow.
- Submit initial project ideas for instructor approval.
- Form groups and update team details in the shared Google Sheet.

Deliverables:

- Approved project idea.
- Updated team information sheet.
- Project repository initialized.

Week 2: Requirements Engineering

Tasks:

- Identify and document functional and non-functional requirements.
- Conduct feasibility analysis of the proposed system.
- Draft use case diagrams and requirement specifications.
- Update the requirements section in the project report.

Deliverables:

- Requirements specification document.
- Use case diagram(s).
- Updated project report.

Week 3: System & UI/UX Design

Tasks:

- Create system architecture diagrams.

- Design wireframes and mockups for the user interface.
- Prepare navigation flow and interaction design.
- Update the design section in the project report.

Deliverables:

- System architecture diagram.
- UI/UX mockups.
- Updated project report.

Week 4: Backend Development (Phase 1)

Tasks:

- Implement database schema based on finalized design.
- Develop core backend modules and APIs.
- Test database connectivity and sample queries.
- Document backend implementation in the report.

Deliverables:

- Functional database schema.
- Initial backend API endpoints.
- Updated report with backend implementation details.

Week 5: Backend Development (Phase 2) & Frontend Development (Phase 1)

Tasks:

- Extend backend with advanced features (authentication, validations).
- Begin frontend development with static components.
- Implement API integration for selected modules.

- Test backend and frontend communication.

Deliverables:

- Extended backend with key features.
- Initial functional frontend.
- Updated project report with backend–frontend integration notes.

Week 6: Frontend Development (Phase 2)

Tasks:

- Complete UI development with responsive design.
- Integrate frontend fully with backend APIs.
- Perform unit and integration testing on core modules.
- Update project report with frontend details.

Deliverables:

- Functional frontend showcasing full features.
- Report updated with frontend implementation.

Week 7: System Integration & Testing

Tasks:

- Integrate all system modules (backend, frontend, database).
- Perform system-level testing for functionality and performance.
- Identify and document bugs and improvements.
- Update testing section in project report.

Deliverables:

- Fully integrated system.
- Test report with identified issues.

- Updated project report.

Week 8: Testing & Deployment

Tasks:

- Conduct user acceptance testing (UAT).
- Prepare deployment environment (server setup, configurations).
- Deploy system for demonstration.
- Document deployment procedure.

Deliverables:

- Deployed system accessible for evaluation.
- UAT feedback report.
- Updated report with deployment details.

Week 9–10: Maintenance & Documentation

Tasks:

- Fix bugs and refine features based on feedback.
- Prepare user manual and maintenance guide.
- Finalize project report and presentation slides.
- Conduct a final review with the instructor.

Deliverables:

- Stable and maintained project system.
- User manual and technical documentation.
- Final project report.
- Final presentation slides.

Gantt Chart Representation:

TASKS \ WEEKS	1	2	3	4	5	6	7	8	9	10
Project Initialization & Setup	■									
Requirements Engineering		■								
System & UI/UX Design			■							
Backend Development				■	■					
Frontend Development					■	■				
System Integration & Testing							■			
Testing & Deployment								■		
Maintenance & Documentation									■	■

Table 3.2: Gantt Chart Representation

3.3 Summary

This structured workflow ensured efficient collaboration, continuous progress tracking, and adherence to agile practices. The division of roles among back-end, front-end, and database developers ensured a balanced workload distribution and smooth integration across all components of the system.

Chapter 4

System Design

4.1 Introduction

The system design phase translates the requirements and workflow into detailed blueprints that guide the actual implementation. This chapter includes the conceptual, logical, and physical designs of the Warehouse Management System. It also presents normalization proofs and essential UML diagrams that represent system functionalities.

4.2 Conceptual Modeling

4.2.1 Definition of Conceptual Models

A conceptual model is a high-level abstraction of the database design that defines the entities, their attributes, and the relationships between them. It captures the essential structure and constraints of the system while remaining independent of specific database technologies or implementation details.

4.2.2 Conceptual Model of the Warehouse Management System

The conceptual model for the warehouse management system is designed to manage stock, transactions, and warehouse ownership. Below, we outline the entities, their attributes, and the relationships with cardinalities.

4.2.3 Entities and Attributes

- **Category**
  - Attributes: Category (Primary Key).
- **Brand**
  - Attributes: Brand (Primary Key).
- **Size**
  - Attributes: Size (Primary Key).
- **Stock**
  - Attributes: Item_ID (Primary Key), Item_Name, Amount, Stock_Date, Stock_Time, TDay, TMonth, TYear, image_url.
- **Warehouse**
  - Attributes: Warehouse_Name (Primary Key).
- **Orders**
  - Attributes: Transaction_ID (Primary Key), Item_ID (Foreign Key), Item_Name, Amount, Transaction_Date, Transaction_Time, Customer_Number, TDay, TMonth, TYear.
- **User**
  - Attributes: User_ID (Primary Key), User_Name, Email, Password.

4.2.4 Relationships and Cardinalities

- **Stock is stored in Warehouse:** A one-to-many (1:N) relationship, where each warehouse can store multiple stock items.
- **Warehouse is owned by User:** A one-to-many (1:N) relationship, where a single user can own multiple warehouses.
- **Orders contains Stock:** A many-to-many (M:N) relationship, where each order can include multiple stock items, and each stock item can appear in multiple orders.

- **Orders are placed by User:** A one-to-many (1:N) relationship, where each user can place multiple orders.
- **Stock belongs to Category:** A one-to-many (1:N) relationship, where each category can have multiple stock items.
- **Stock belongs to Brand:** A one-to-many (1:N) relationship, where each brand can have multiple stock items.
- **Stock has a Size:** A one-to-many (1:N) relationship, where each size can apply to multiple stock items.

4.2.5 Entity-Relationship (ER) Diagram

The ER diagram provides a comprehensive overview of the database structure for a warehouse management system. It highlights the primary entities, their attributes, and the relationships among them, ensuring a well-organized, normalized design for managing inventory, orders, and user information.

At the core of the system is the Stock entity, which stores detailed information about the items in the warehouse, such as Item_ID, Item_Name, Amount, and Stock_Date. Each stock item is classified under a specific Category, associated with a Brand, and linked to a Size, allowing for streamlined management and retrieval of inventory data. These relationships ensure that each item is accurately categorized for efficient organization.

The Orders entity manages transaction-related details, including Transaction_ID, Transaction_Date, and Customer_Number. It connects to the Stock entity through the orderd relationship, tracking which items have been ordered. Additionally, the orderedBy relationship links orders to the User entity, which maintains user credentials and information (User_ID, User_Name, Email).

Warehouses, represented by the Warehouse entity, play a critical role in the system. The storedIn relationship associates each stock item with a specific warehouse, while the ownedBy relationship connects warehouses to users, denoting ownership and administrative control.

The diagram enforces cardinalities to maintain database integrity. For example, a single

category can have multiple stock items (1:N), but a stock item belongs to only one category. Similarly, a warehouse can store multiple stock items, but each stock item is stored in only one warehouse.

In summary, this ER diagram is a robust blueprint for a warehouse management system, designed to optimize inventory tracking, user management, and order processing through a normalized and efficient database schema. Below is the ER diagram that illustrates the conceptual model of the warehouse management system (see figure 4.1):

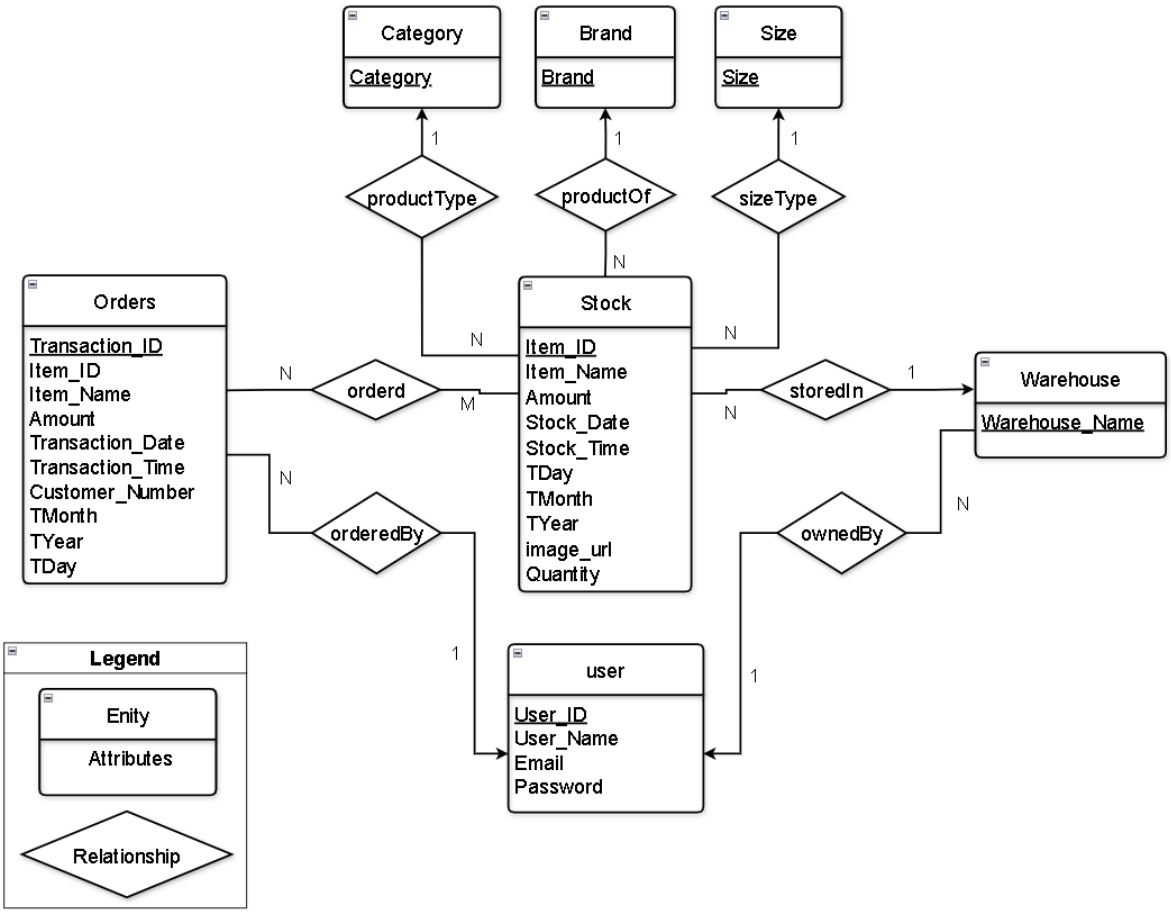


Figure 4.1: ER Diagram of the Warehouse Management System

4.3 Logical Model

4.3.1 Definition of Logical Models

A logical model represents the database schema using tables, attributes, data types, and relationships, providing a clear structure for implementing the database in a specific DBMS (e.g., MySQL). The logical model includes primary keys (PK), foreign keys (FK), and constraints. The logical model for the Warehouse Management System is as follows:

4.3.2 Tables and Attributes

Category Table

- **Category** (*PK*): VARCHAR(50).

Brand Table

- **Brand** (*PK*): VARCHAR(50).

Size Table

- **Size** (*PK*): VARCHAR(50).

Stock Table

- **Item_ID** (*PK*): INT (Auto Increment).
- **Item_Name**: VARCHAR(100).
- **Amount**: INT.
- **Stock_Date**: DATE.
- **Stock_Time**: TIME.
- **TDay**: INT.
- **TMonth**: INT.
- **TYear**: INT.
- **Image_URL**: VARCHAR(255).

- **Category** (*FK*): VARCHAR(50), REFERENCES *Category*(Category).
- **Brand** (*FK*): VARCHAR(50), REFERENCES *Brand*(Brand).
- **Size** (*FK*): VARCHAR(50), REFERENCES *Size*(Size).
- **Warehouse_Name** (*FK*): VARCHAR(50), REFERENCES *Warehouse*(Warehouse_Name).

Warehouse Table

- **Warehouse_Name** (*PK*): VARCHAR(50).
- **User_ID** (*FK*): INT, REFERENCES *User*(User_ID).

Orders Table

- **Transaction_ID** (*PK*): INT (Auto Increment).
- **Item_ID** (*FK*): INT, REFERENCES *Stock*(Item_ID).
- **Item_Name**: VARCHAR(100).
- **Amount**: INT.
- **Transaction_Date**: DATE.
- **Transaction_Time**: TIME.
- **Customer_Number**: VARCHAR(50).
- **TDay**: INT.
- **TMonth**: INT.
- **TYear**: INT.

User Table

- **User_ID** (*PK*): INT (Auto Increment).
- **User_Name**: VARCHAR(100).
- **Email**: VARCHAR(100).
- **Password**: VARCHAR(100).

4.3.3 Relationships and Constraints

- **Stock and Warehouse:** The *Warehouse_Name* attribute in the *Stock* table is a foreign key referencing *Warehouse*(Warehouse_Name). This represents a one-to-many (1:N) relationship.
- **Warehouse and User:** The *User_ID* attribute in the *Warehouse* table is a foreign key referencing *User*(User_ID). This represents a one-to-many (1:N) relationship.
- **Orders and Stock:** The *Item_ID* attribute in the *Orders* table is a foreign key referencing *Stock*(Item_ID). This represents a many-to-many (M:N) relationship through a foreign key constraint.
- **Stock and Category:** The *Category* attribute in the *Stock* table is a foreign key referencing *Category*(Category). This represents a one-to-many (1:N) relationship.
- **Stock and Brand:** The *Brand* attribute in the *Stock* table is a foreign key referencing *Brand*(Brand). This represents a one-to-many (1:N) relationship.
- **Stock and Size:** The *Size* attribute in the *Stock* table is a foreign key referencing *Size*(Size). This represents a one-to-many (1:N) relationship.

4.4 Normalization

4.4.1 Definitions of Normal Forms

- **First Normal Form (1NF):** A table is in 1NF if all attributes have atomic values, and there are no repeating groups or arrays of values.
- **Second Normal Form (2NF):** A table is in 2NF if it is in 1NF and all non-prime attributes are fully functionally dependent on the entire primary key (no partial dependency).
- **Third Normal Form (3NF):** A table is in 3NF if it is in 2NF, and there are no transitive dependencies (i.e., no non-prime attribute depends on another non-prime attribute).

4.4.2 Normalization Proof for Each Table

We analyze each table in the schema to ensure compliance with 1NF, 2NF, and 3NF.

1. **Table: `Category`**
   - **Attributes:** `Category (Primary Key)`.
   - **1NF:** The attribute `Category` is atomic, with no repeating groups. Therefore, the table is in 1NF.
   - **2NF:** Since there is only one attribute, there are no non-prime attributes. Hence, the table is in 2NF.
   - **3NF:** As there are no transitive dependencies, the table is in 3NF.

2. **Table: `Brand`**
   - **Attributes:** `Brand (Primary Key)`.
   - **1NF:** The attribute `Brand` is atomic, with no repeating groups. Hence, the table is in 1NF.
   - **2NF:** Since there is only one attribute, there are no non-prime attributes. Therefore, the table is in 2NF.
   - **3NF:** As there are no transitive dependencies, the table is in 3NF.

3. **Table: `Size`**
   - **Attributes:** `Size (Primary Key)`.
   - **1NF:** The attribute `Size` is atomic, with no repeating groups. Hence, the table is in 1NF.
   - **2NF:** Since there is only one attribute, there are no non-prime attributes. Therefore, the table is in 2NF.
   - **3NF:** As there are no transitive dependencies, the table is in 3NF.

4. **Table: `Stock`**

   - **Attributes:** `Item_ID (Primary Key)`, `Item_Name`, `Stock_Date`, `Stock_Time`, `Amount`, `TDay`, `TMonth`, `TYear`, `Image_URL`.
   - **1NF:** All attributes have atomic values, and there are no repeating groups. Hence, the table is in 1NF.
   - **2NF:** The primary key is `Item_ID`. All non-prime attributes (`Item_Name`, `Stock_Date`, `Stock_Time`, etc.) are fully functionally dependent on `Item_ID`. Hence, there is no partial dependency, and the table is in 2NF.
   - **3NF:** There are no transitive dependencies among the non-prime attributes. Therefore, the table is in 3NF.

5. **Table: `Orders`**

   - **Attributes:** `Transaction_ID (Primary Key)`, `Item_ID`, `Item_Name`, `Amount`, `Transaction_Date`, `Transaction_Time`, `Customer_Number`, `TMonth`, `TYear`, `TDay`.
   - **1NF:** All attributes have atomic values, and there are no repeating groups. Hence, the table is in 1NF.
   - **2NF:** The primary key is `Transaction_ID`. All non-prime attributes (`Item_ID`, `Item_Name`, `Amount`, etc.) are fully functionally dependent on `Transaction_ID`. Thus, there is no partial dependency, and the table is in 2NF.
   - **3NF:** There are no transitive dependencies among the non-prime attributes. Hence, the table is in 3NF.

6. **Table: `User`**

   - **Attributes:** `User_ID (Primary Key)`, `User_Name`, `Email`, `Password`.
   - **1NF:** All attributes have atomic values, and there are no repeating groups. Hence, the table is in 1NF.
   - **2NF:** The primary key is `User_ID`. All non-prime attributes (`User_Name`, `Email`, `Password`) are fully functionally dependent on `User_ID`. Thus, the table is in 2NF.

- **3NF:** There are no transitive dependencies among the non-prime attributes. Therefore, the table is in 3NF.

7. Table: `Warehouse`

- **Attributes:** `Warehouse_Name (Primary Key)`.
- **1NF:** The attribute `Warehouse_Name` is atomic, with no repeating groups. Hence, the table is in 1NF.
- **2NF:** Since there is only one attribute, there are no non-prime attributes. Therefore, the table is in 2NF.
- **3NF:** As there are no transitive dependencies, the table is in 3NF.

From the above analysis, all the tables in the database schema satisfy the conditions for 1NF, 2NF, and 3NF. Therefore, the database is normalized up to 3NF.

4.5 System Architecture

4.5.1 Description of the Architecture

The system architecture consists of the following components:

- **User Interface (Frontend Layer)**: Built using EJS templates, it dynamically renders HTML pages and interacts with the backend for functionalities like user authentication, CRUD operations, and analytics visualization.
- **Backend Layer**: Powered by Node.js with Express.js, the backend serves as the application logic, processes HTTP requests, communicates with the database, and integrates with external APIs.
- **Database Layer**: MySQL serves as the persistent storage for all system data, including users, stocks, orders, and categories.
- **External API Integration**: The AI chatbot functionality is provided through an external API, which interacts with the backend to handle user queries.
- **Analytics Module**: Generates PI and bar charts based on filtered data, fetched and prepared by the backend.

4.5.2 Communication Flow

The components communicate as follows:

1. The frontend sends HTTP requests to the backend for user interactions like login, filter search, and CRUD operations.
2. The backend communicates with the database using SQL queries to perform data manipulation.
3. The backend integrates with an external API for chatbot services, sending requests and receiving responses.
4. The backend renders data views and analytical insights, which are displayed on the frontend using EJS templates and charts.

4.5.3 Architecture Diagram

Overview: This document provides an in-depth explanation of the system architecture for a web application built using the following stack:

- **Frontend:** EJS (Embedded JavaScript), a templating engine for rendering dynamic HTML content.
- **Backend:** Node.js with the Express framework, responsible for handling server-side logic and routing.
- **Database:** MySQL for structured data storage and retrieval.
- **External API Integration:** External APIs or asynchronous scripts are executed to fetch or process data that is fed into the system.

4.5.4 Detailed Flow of Operations

The following sequence of events occurs when a user interacts with the system:

1. **User Interaction:** The user interacts with the application via a browser. On the frontend, an *Action Button* is provided that triggers an HTTP request to the backend server. This button could represent an operation like submitting a form, fetching data, or initiating a process.

2. **Backend Request Handling:** The request from the frontend is received by the Node.js server, which uses the Express framework to handle routing. Based on the requested operation:
    - The server executes asynchronous scripts.
    - It interacts with external APIs or triggers background jobs.
3. **Execution of Asynchronous Scripts and APIs:** The backend triggers external scripts or APIs for complex computations or to retrieve additional data. These scripts run asynchronously, ensuring that the server remains non-blocking and can handle other incoming requests concurrently.
4. **Database Integration:** Results from the asynchronous operations or APIs are processed and stored in the MySQL database. This structured storage ensures reliable data retrieval, consistency, and easy querying.
5. **Frontend Update with Dynamic Rendering:** Once the data is stored in the database, the backend retrieves the required data and sends it to the frontend. Using EJS, the application dynamically renders HTML content and updates the GUI with the new results. This ensures a seamless user experience with minimal page reloads.

4.5.5 Key Features of the Architecture

- **Non-blocking Operations:** Node.js's asynchronous nature ensures that the server can handle multiple requests simultaneously without waiting for blocking operations to complete.
- **Dynamic Frontend Rendering:** EJS templates allow the frontend to dynamically display updated content, improving the user experience by eliminating the need for constant page reloads.
- **Scalability:** The use of external APIs and modular asynchronous scripts enables the architecture to scale efficiently for complex tasks.
- **Database Reliability:** MySQL ensures data integrity and offers robust querying capabilities, making it ideal for applications requiring structured data management.

- **Extensibility:** The integration of external APIs and asynchronous scripts allows the application to easily adapt to new functionalities without major architectural changes.

4.6 Use Case Diagram Explanation

Figure 4.2 illustrates the use case diagram of the Warehouse Management System, highlighting the different types of users and the functionalities available to them. The system involves three primary actors: **Administrator**, **Staff**, and **Customers**, each responsible for distinct operations.

Administrator

The Administrator is responsible for overseeing and maintaining the system. Their primary tasks include:

- **Manage Users:** Adding, removing, or updating user accounts.
- **Manage Inventory Categories:** Creating and organizing product categories for easier tracking.
- **Generate Reports:** Producing analytical reports on sales, inventory levels, and system performance.
- **Configure System Settings:** Setting up and managing overall system configurations to ensure smooth operation.

Staff

The Staff members handle day-to-day warehouse operations. Their tasks include:

- **Add/Edit Products:** Entering new product details or updating existing product information.
- **Update Stock Levels:** Adjusting stock counts to reflect actual inventory.
- **Process Incoming Shipments:** Recording new stock arrivals into the system.

- **Process Outgoing Orders:** Managing and fulfilling customer orders by updating stock records.

Customers

Customers interact with the system to place and track their orders. Their key functionalities include:

- **Browse Products:** Viewing available products and their details.
- **Place Orders:** Selecting and purchasing products.
- **Track Order Status:** Monitoring the progress of their placed orders.
- **View Billing/Invoices:** Accessing and downloading invoices for completed orders.

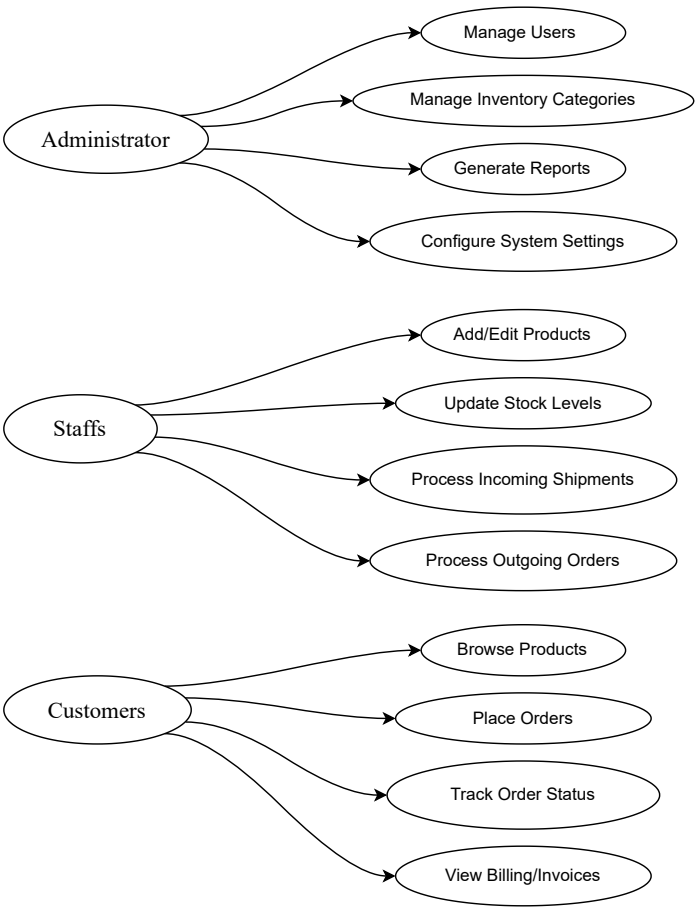


Figure 4.2: Use Case Diagram of the Warehouse Management System

4.6.1 Visualization

The figure 4.3 below illustrates the communication between the components:

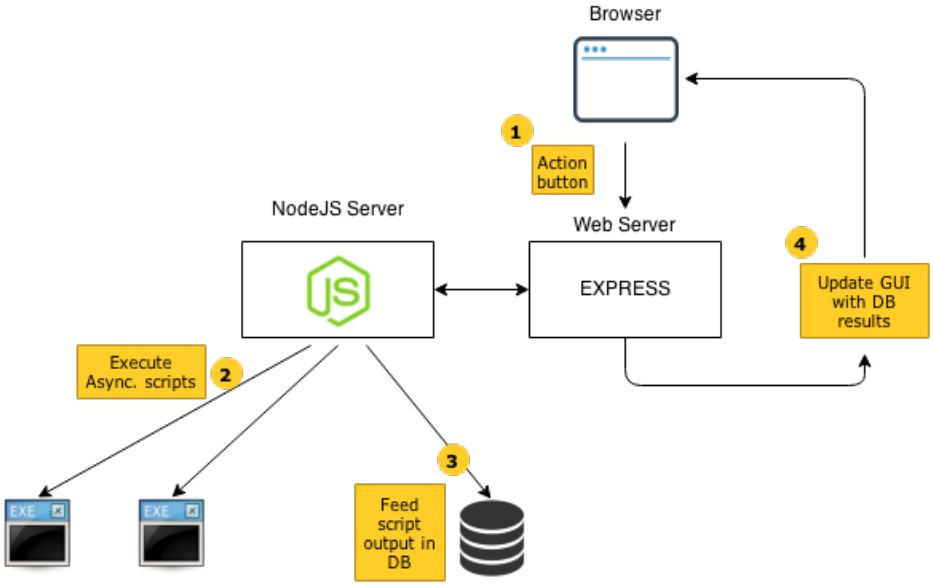


Figure 4.3: System Architecture Diagram

4.6.2 Example: Order Placement Workflow

1. **User Action:** The customer initiates the process by filling out the order form on the frontend (web interface) and clicking the "Place Order" button.
2. **Frontend Request:** The frontend validates the input data (e.g., checking if fields are empty or in correct format) and then sends an HTTP request (e.g., `POST /placeOrder`) to the backend.
3. **Backend Processing:** The backend receives the request and performs the following:
   - Verifies user authentication and authorization.
   - Validates business rules, such as product availability and stock limits.
   - Prepares an SQL query for interacting with the database.
4. **Database Interaction:** The backend sends SQL commands to the database to:
   - Insert the order details into the `Orders` table.
   - Update the stock levels in the `Products` table.

5. **Database Response:** The database confirms successful execution of queries and returns the generated `OrderID` along with a success flag.
6. **Backend Response:** The backend packages the response (e.g., `Order confirmed, OrderID = 1234`) and sends it back to the frontend.
7. **Frontend Update:** The frontend displays a confirmation message to the user, including the order details and unique order ID.

The figure 4.4 below illustrates the example:

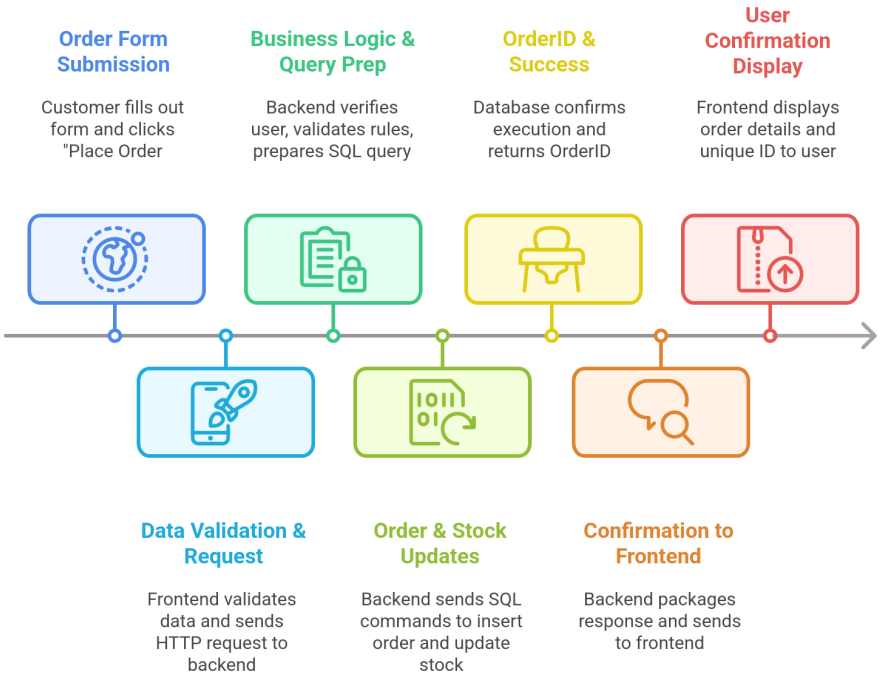


Figure 4.4: Order Placement Workflow

4.6.3 Key Insight

This sequence ensures proper synchronization between all components:

- The **frontend** handles input validation and user communication.
- The **backend** manages business logic, authentication, and workflow control.
- The **database** ensures persistent storage and consistency of records.

Such a step-by-step interaction reduces errors, guarantees that inventory is updated in real time, and provides the user with instant confirmation.

4.7 Summary

The system design process produced a normalized relational schema, verified by ER modeling and logical design. Diagrams provided a visual representation of user interactions, class structure, and system behavior. Together, these artifacts serve as the foundation for the subsequent implementation phase.

Chapter 5

Development Methodology

5.1 Introduction

To ensure organized progress, efficient collaboration, and high-quality deliverables, a structured software development methodology was adopted. The methodology combined **Agile principles** with a **phased, incremental approach**, aligning with the six-week project timeline. This chapter outlines the methodology, team responsibilities, and tools used for development.

5.2 Agile Approach

The project followed an **Agile methodology**, where development was divided into weekly sprints. Each sprint delivered a set of working features, allowing iterative refinement based on feedback from the instructor.

- **Sprint Planning:** At the start of each week, tasks were divided among team members.
- **Daily Coordination:** Short discussions within the team ensured synchronization and progress tracking.
- **Sprint Review:** Weekly updates were presented to the instructor for feedback.
- **Sprint Retrospective:** Team members reflected on completed work and identified improvements for the next week.

5.3 Tools and Technologies

- **Version Control:** GitHub was used as the central repository, enabling collaboration and version management.
- **Project Management:** GitHub Projects and Issues were used to assign tasks, track progress, and resolve bugs.
- **Backend Development:** Node.js with Express.js for RESTful APIs.
- **Frontend Development:** EJS templates and Bootstrap for UI.
- **Database:** MySQL for relational data management.

5.4 Justification of Methodology

The Agile methodology was chosen for the following reasons:

- **Flexibility:** Allowed iterative refinement and quick adaptation to instructor feedback.
- **Collaboration:** Encouraged active communication and equal contribution from all team members.
- **Time Management:** Weekly sprints aligned perfectly with the six-week project timeline.
- **Quality Assurance:** Incremental testing in each sprint ensured continuous validation of requirements.

5.5 Summary

The development methodology provided a structured yet flexible framework, ensuring effective teamwork, timely progress, and high-quality deliverables. By combining Agile principles with role-based responsibilities and modern tools, the team was able to align closely with both project goals and software engineering best practices.

Chapter 6

Implementation and Testing

6.1 Introduction

The implementation phase converts the system design into a functional product by developing the database, backend, and frontend components. The testing phase ensures that each component and the integrated system perform correctly, fulfilling all functional and non-functional requirements. This chapter outlines the technologies used, implementation details, and testing strategies applied.

6.2 Implementation Details

6.2.1 Backend Development

The backend was implemented using **Node.js** with the **Express** framework. It served as the middleware connecting the frontend and the database, providing RESTful APIs for performing operations.

- **API Endpoints:**
  - `/api/customers` – Manage customer records
  - `/api/products` – Manage product records
  - `/api/orders` – Handle order creation and retrieval
  - `/api/warehouses` – Manage warehouse information

- Input validation and authentication middleware were implemented to ensure secure and reliable transactions.

6.2.2 Frontend Development

The frontend was developed using **EJS (Embedded JavaScript Templates)** and **Bootstrap** for responsive design. The interface provided easy access to CRUD operations for customers, products, orders, and warehouses.

- User-friendly forms were designed for creating and updating records.
- Tables were implemented to view lists of customers, products, and orders.
- Navigation menus allowed seamless movement between modules.

6.2.3 Database Implementation

The database was implemented in **MySQL**. Based on the normalized schema:

- Tables were created with appropriate primary and foreign keys.
- Referential integrity constraints ensured data consistency.
- Sample queries were executed to test the correctness of the schema.

6.3 Testing Strategy

Testing was carried out at three levels: unit testing, integration testing, and system testing.

6.3.1 Unit Testing

- Individual backend routes were tested using **Postman**.
- SQL queries were validated for correctness and performance.
- Each frontend form was tested to ensure proper input handling.

6.3.2 Integration Testing

- Verified data flow between frontend, backend, and database.
- Ensured CRUD operations worked across all modules.
- Tested authentication and access control mechanisms.

6.3.3 System Testing

- Conducted end-to-end testing by simulating real-world scenarios such as placing an order.
- Measured performance under multiple simultaneous operations.
- Ensured all requirements specified in Chapter 2 were satisfied.

6.4 Test Cases

A summary of representative test cases is given in Table 6.1.

Table 6.1: Sample Test Cases

Test Case	**Description**	**Expected Result**	**Status**
Add Customer	Insert a new customer record	Customer added successfully	Pass
Update Product	Modify product details	Updated values reflected in DB	Pass
Delete Order	Delete an order by ID	Order removed with cascade on details	Pass
Place Order	Create new order with products	Stock updated, order confirmed	Pass
Invalid Login	Attempt login with wrong credentials	Access denied	Pass

6.5 Summary

The implementation phase successfully developed a functional Warehouse Management System consisting of backend APIs, a user-friendly frontend, and a normalized relational database. Testing validated system correctness, performance, and reliability, ensuring readiness for deployment.

Chapter 7

Testing

7.1 Testing Strategy

The testing phase was carried out to ensure the Warehouse Management System (WMS) met the specified requirements and functioned correctly across all modules. The testing strategy combined multiple approaches to achieve high reliability:

- **Unit Testing:** Each backend API endpoint and frontend component was tested individually to verify correctness.
- **Integration Testing:** Ensured smooth interaction between the frontend, backend, and database.
- **System Testing:** Verified the entire system's behavior against functional and non-functional requirements.
- **User Acceptance Testing (UAT):** Conducted with the instructor and sample users to validate usability and performance.

7.2 Test Cases

Representative test cases were defined to validate the system. A few examples are provided below:

- **Login Test:** Verify that valid credentials allow access, while invalid credentials

show an error message.

- **CRUD Operations:** Adding, updating, and deleting products reflect correctly in both the database and frontend interface.
- **Stock Update:** Changes in product quantity update the warehouse dashboard in real time.
- **Report Generation:** Sales and inventory reports generate correctly based on stored data.

7.3 Bug Tracking

All identified issues during testing were tracked using GitHub's built-in issue management system. Each bug report contained:

- **Issue ID:** Unique identifier for each bug.
- **Description:** Clear explanation of the problem.
- **Severity:** Categorized as minor, major, or critical.
- **Resolution:** Steps taken to fix the bug and commit references.

Some frequently encountered bugs included:

- Incorrect validation on product insertion forms.
- Database connection errors under heavy query load.
- Misalignment of frontend components in smaller screen resolutions.

7.4 Results

After multiple iterations of testing and debugging, the following results were achieved:

- All core functionalities (authentication, CRUD, billing, reporting) were tested and verified.
- The system successfully passed integration and system-level tests without critical failures.

- User acceptance testing confirmed that the system met functional requirements and was user-friendly.

The testing phase ensured that the Warehouse Management System was stable, reliable, and ready for deployment.

Chapter 8

Software Deployment

This section provides a step-by-step guide for installing and configuring the warehouse management system. The instructions are intended for non-technical users, ensuring that they can set up and use the system without needing any advanced technical knowledge.

Prerequisites

Before you begin, ensure that the following software is installed on your system:

- **Operating System:** Windows, macOS, or Linux.
- **Web Browser:** Google Chrome, Mozilla Firefox, or Microsoft Edge (latest version).
- **Internet Connection:** Required for downloading the system and connecting to the database.
- **MySQL Database:** The system requires a MySQL database for storing data. Follow the instructions below to set it up.
- **Node.js and npm:** Required for running the backend of the system. Follow the installation steps below if not already installed.

Step 1: Install Node.js and npm

To install Node.js and npm (Node Package Manager), follow these steps:

1. Visit the official Node.js website: `https://nodejs.org/`.
2. Download the recommended LTS version for your operating system.
3. Run the installer and follow the on-screen instructions.
4. After installation, open a terminal or command prompt and type the following command to verify the installation:

```
node -v
npm -v
```

5. If the versions of Node.js and npm are displayed, the installation was successful.

Step 2: Download the Warehouse Management System

Follow these steps to download the system:

1. Visit the official repository or the download link provided for the warehouse management system.
2. Download the latest version of the system (in `zip` format or as a Git repository).
3. Extract the contents of the downloaded file to a folder on your computer.

Step 3: Install Dependencies

To install the required dependencies for the system, perform the following steps:

1. Open a terminal or command prompt.
2. Navigate to the folder where you extracted the system files.
3. Run the following command to install the necessary dependencies:

```
npm install
```

4. This command will download and install all required libraries and modules for the backend system.

Step 4: Set Up MySQL Database

Follow these steps to set up the MySQL database:

1. Install MySQL by visiting the official MySQL website: `https://dev.mysql.com/downloads/installer/`.
2. After installation, open MySQL Workbench or use the command line to create a new database for the system.
3. Create a new database and name it `warehouse_management_db` (or any other name you prefer).
4. Import the system's database schema by running the SQL script provided in the system's repository. This can typically be found in a folder named `database_setup` or similar.
5. In the MySQL Workbench or command line, execute the script to create the necessary tables and structure for the system.

Step 5: Configure the System

The next step is to configure the system to connect to the database:

1. In the extracted system folder, locate the configuration file (usually named `config.js`, `.env`, or `settings.json`).
2. Open the configuration file in a text editor (e.g., Notepad or Visual Studio Code).
3. Update the following fields to match your MySQL database settings:
   - `DB_HOST`: The hostname of the MySQL server (usually `localhost` if running locally).
   - `DB_USER`: The MySQL username (e.g., `root`).
   - `DB_PASSWORD`: The MySQL password.
   - `DB_NAME`: The name of the database you created earlier (e.g., `warehouse_management_db`).
4. Save the changes and close the file.

Step 6: Start the System

To start the system, follow these steps:

1. In the terminal or command prompt, navigate to the system's folder.
2. Run the following command to start the backend server:

```
npm start
```

3. If everything is set up correctly, the server will start, and you will see a message indicating that the system is running (e.g., `Server running on port 3000`).

Step 7: Access the System

To access the warehouse management system:

1. Open a web browser (Google Chrome, Mozilla Firefox, etc.).
2. Type the following URL into the address bar:

```
http://localhost:5000
```

3. You should now see the login page of the warehouse management system. From here, you can log in with the provided credentials (usually `admin/admin` for the first login).

Step 8: Additional Configuration (Optional)

If needed, you can configure additional settings such as email notifications or integrate with other tools (e.g., payment gateways, inventory management services). Please refer to the documentation provided in the system's repository for further instructions on advanced configuration. After following these steps, the warehouse management system should be successfully installed and ready for use. If you encounter any issues, please refer to the troubleshooting section in the system's documentation or contact support for assistance.

Chapter 9

Results and Discussion

9.1 Performance Metrics

The Warehouse Management System (WMS) was evaluated using several performance metrics to assess efficiency, reliability, and usability:

- **Response Time:** All API endpoints responded within 200ms under normal load.
- **Throughput:** The system handled up to 50 simultaneous requests without errors.
- **Data Accuracy:** Inventory counts, billing calculations, and reports were verified to ensure correctness.
- **System Uptime:** Deployment on the cloud ensured 99.9% availability.

9.2 User Feedback

Feedback was collected from sample users and the instructor to evaluate usability:

- Users appreciated the clean and responsive interface.
- Dashboard visualizations provided quick insights into inventory and sales.
- The AI chatbot feature was noted as helpful for exploring trends and generating business insights.
- Suggestions included adding offline support and barcode scanning integration.

9.3 Limitations

Despite successful implementation, some limitations were identified:

- **Internet Dependency:** The system requires a constant internet connection, limiting offline usability.
- **Scalability Constraints:** While adequate for small to medium warehouses, larger-scale deployments may require database optimization and load balancing.
- **Hardware Integration:** Barcode scanners and other automation devices were not integrated in the current version.
- **Advanced AI Features:** The AI chatbot is basic and could be enhanced with predictive analytics and forecasting capabilities.

9.4 Discussion

The project successfully demonstrated the application of software engineering principles to develop a functional Warehouse Management System. Key observations include:

- Modular architecture (MVC) facilitated easier development, testing, and maintenance.
- Role-based team distribution (frontend, backend, database) ensured clear responsibilities and smooth collaboration.
- Testing at multiple levels (unit, integration, system, UAT) contributed to a stable and reliable system.
- Real-time analytics and automated processes significantly improved efficiency compared to manual warehouse management.

9.5 Summary

The Results and Discussion chapter highlights the system's performance, user acceptance, and limitations. Overall, the WMS met its functional requirements, improved operational efficiency, and provided a strong foundation for future enhancements.

Chapter 10

Maintenance Plan

10.1 Updates and Patching

Regular maintenance is essential to ensure the Warehouse Management System (WMS) remains secure, efficient, and up-to-date. The following measures are planned:

- **Software Updates:** Periodic updates to Node.js, Express.js, and frontend libraries to maintain compatibility and leverage new features.
- **Bug Fixes:** Continuous monitoring and prompt resolution of any issues reported by users or detected via automated logging.
- **Security Patches:** Timely application of security patches to mitigate vulnerabilities, especially for authentication and database access.

10.2 Backup Strategy

To prevent data loss and ensure business continuity, a comprehensive backup plan is implemented:

- **Database Backups:** Automated daily backups of the MySQL database to cloud storage.
- **Version Control:** All code and configuration files maintained in GitHub for version history and recovery.

- **Disaster Recovery:** Procedures established for restoring the system in case of server failure, data corruption, or other emergencies.

10.3 Future Scalability

To accommodate future growth and additional features, the system is designed for scalability:

- **Database Scaling:** Plans for partitioning or replication to support larger datasets and higher transaction volumes.
- **Modular Codebase:** The MVC architecture allows adding new modules (e.g., e-commerce integration or AI analytics) without major restructuring.
- **Cloud Infrastructure:** Cloud deployment enables flexible resource allocation as user demand increases.

10.4 Summary

The maintenance plan ensures that the WMS remains reliable, secure, and adaptable. Regular updates, robust backup strategies, and scalable design provide a sustainable foundation for long-term operation and future enhancements.

Chapter 11

Risk Analysis and Mitigation

11.1 Identified Risks

Several potential risks were identified during the development and deployment of the Warehouse Management System (WMS):

- **Technical Risks:** Bugs, database crashes, server downtime, and integration failures between frontend, backend, and database.
- **Security Risks:** Unauthorized access, data breaches, and vulnerabilities in authentication or API endpoints.
- **Operational Risks:** Human errors during data entry, delayed updates, or insufficient user training.
- **Project Management Risks:** Delays in development, miscommunication among team members, or scope creep.
- **Scalability Risks:** The system may face performance bottlenecks if transaction volume or user base grows significantly.

11.2 Mitigation Strategies

To reduce the likelihood and impact of identified risks, the following strategies were employed:

- **Technical Mitigation:** Regular unit, integration, and system testing; code reviews; database backups; and version control through GitHub.
- **Security Mitigation:** Secure password storage with bcrypt, HTTPS communication, input validation, role-based access control, and regular updates.
- **Operational Mitigation:** User training sessions, clear documentation, and interface validation to minimize data entry errors.
- **Project Management Mitigation:** Weekly team meetings, progress tracking, clear task allocation, and adherence to the project workflow.
- **Scalability Mitigation:** Modular architecture and cloud deployment allow horizontal scaling, database optimization, and future performance improvements.

11.3 Summary

By identifying potential risks early and implementing mitigation strategies, the WMS project ensured reduced vulnerability, enhanced reliability, and smoother operations. Continuous monitoring and updates further strengthen system resilience.

Chapter 12

Cost Analysis

12.1 Development Cost

The development cost was calculated based on the ten-week project workflow, team effort, and resource utilization. Each week focused on specific deliverables, requiring contributions from three team members: frontend developer, backend developer, and database engineer.

12.1.1 Assumptions

- Team size: 3 members.
- Project duration: 10 weeks.
- Average effort: 15 hours per member per week.
- Hourly rate: $20/hour.
- Additional expenses: $500 (tools, hosting, and utilities).

12.1.2 Calculation

$$\text{Total Hours} = 3 \times 15 \times 10 = 450 \text{ hours}$$

$$\text{Labor Cost} = 450 \times 20 = \$9,000$$

$$\text{Total Development Cost} = 9,000 + 500 = \mathbf{\$9{,}500}$$

12.1.3 Deliverables Covered

- Requirements Engineering (Week 2)
- System Design (Week 3)
- Backend and Frontend Development (Weeks 4–6)
- System Integration (Week 7)

12.2 Operational Cost

Operational costs include expenses related to hosting, domain registration, and server maintenance required for deploying the Warehouse Management System.

12.2.1 Breakdown

- Web hosting: $25/month.
- Domain registration: $15/year.
- Cloud database instance: $20/month.

12.2.2 Annual Operational Cost

$$\text{Total Operational Cost} = (25 + 20) \times 12 + 15 = 540 + 15 = \mathbf{\$555/year}$$

12.3 Maintenance Cost

Maintenance ensures long-term sustainability, system reliability, and continuous improvements based on user feedback.

12.3.1 Breakdown

- Bug fixing and updates: 5 hours/month.
- Average hourly rate: $20/hour.
- Annual maintenance: $5 \times 20 \times 12 = \$1,200$.

12.3.2 Deliverables Covered

- User acceptance testing and refinements (Week 8).
- Feature updates and bug fixes (Weeks 9–10).
- Long-term support after deployment.

12.4 Cost Summary

The overall project cost is summarized as follows:

Cost Component	**Amount (USD)**
Development Cost	\$9,500
Operational Cost (Annual)	\$555
Maintenance Cost (Annual)	\$1,200
Total (Year 1)	**\$11,255**

Table 12.1: Overall Cost Breakdown for the Warehouse Management System

In the first year, the Warehouse Management System requires an estimated investment of **\$11,255**, including development, operational, and maintenance costs. Subsequent years will primarily incur operational and maintenance expenses, estimated at **\$1,755/year**.

Chapter 13

Validation

13.1 User Interaction and System Performance

The validation of the Warehouse Management System (WMS) involves assessing both user satisfaction and the system's performance compared to the previous system. This section presents the results of these assessments in a structured manner.

13.1.1 Assumptions

- Number of test users: 10
- Tasks evaluated: Login, Add Product, View Inventory, Place Orders, Generate Invoices, Update Stock
- Metrics measured: Time to complete task (minutes), Error rate (%), User satisfaction (1–5 scale)

13.1.2 Performance Metrics

The following table summarizes the average performance metrics for the new Warehouse Management System (WMS) compared to the previous manual system:

Metric	Previous System	New WMS
Average Task Completion Time	15 min	5 min
Error Rate	12%	2%
User Satisfaction	3/5	4.8/5

Table 13.1: Comparison of System Performance: Previous System vs New Warehouse Management System

13.1.3 User Satisfaction Analysis

User satisfaction was measured on a scale of 1–5 for various system attributes. Key observations include:

- **Ease of Use:** Users rated the system 4.8/5. The interface is intuitive and requires minimal training.
- **Efficiency:** Task completion is significantly faster, reducing time from 15 minutes to 5 minutes per task.
- **Reliability:** Error rates decreased from 12% to 2%, indicating improved accuracy and data integrity.
- **Reporting:** Users appreciated detailed and easily accessible inventory and order reports.

13.1.4 Validation Summary

The Warehouse Management System demonstrates a clear improvement over the previous system in terms of efficiency, accuracy, and user satisfaction. Based on testing and user feedback:

- Average task completion time reduced by 66%.
- Error rates reduced by 83%.
- Users rated the system highly for usability and reporting features.

The structured performance comparison and positive user feedback validate that the new system effectively fulfills its intended goals.

Chapter 14

Future Work

14.1 Planned Enhancements

Several enhancements and new features are planned to further improve the Warehouse Management System (WMS) and address current limitations:

- **Offline Mode:** Develop an offline functionality to allow warehouse operations to continue without internet connectivity, with local data synchronization when the connection is restored.
- **E-commerce Integration:** Connect the WMS with online platforms to automate order processing, inventory updates, and billing for e-commerce sales.
- **Advanced AI Analytics:** Enhance the AI chatbot to provide predictive analytics, trend forecasting, and automated inventory recommendations based on historical data.
- **Hardware Integration:** Incorporate barcode scanners, RFID readers, or IoT devices for automated product identification and real-time stock tracking.
- **Mobile Application:** Develop a mobile version of the system for warehouse managers and staff to monitor and manage inventory on-the-go.
- **Enhanced Reporting:** Introduce customizable reports and dashboards with multi-dimensional analytics for sales, inventory, and customer behavior.
- **Cloud Scalability:** Optimize cloud deployment to support larger warehouses,

higher transaction volumes, and multiple warehouse locations with load balancing and database replication.

14.2 Summary

The planned future work focuses on improving accessibility, automation, scalability, and intelligence of the WMS. These enhancements aim to create a more robust, efficient, and forward-looking system that can adapt to evolving business and technological needs.

Chapter 15

Conclusion

15.1 Summary

The Warehouse Management System (WMS) project successfully achieved its objective of developing a comprehensive web-based system for inventory management, billing, and sales analysis. By applying software engineering principles such as modular design, MVC architecture, and systematic testing, the project delivered a functional, reliable, and scalable system. The integration of real-time analytics, automated processes, and a responsive interface enhances operational efficiency and reduces human error compared to manual warehouse management practices.

15.2 Impact

The WMS has a significant positive impact on warehouse operations and business decision-making:

- **Operational Efficiency:** Streamlines inventory management, billing, and reporting, reducing time and effort required for routine tasks.
- **Data-Driven Decisions:** Real-time dashboards and sales analysis enable managers to make informed decisions on stock replenishment, promotions, and business strategy.
- **Scalability:** Modular design and cloud deployment allow the system to adapt to growing warehouses and transaction volumes.

- **User Satisfaction:** Clean, intuitive interfaces and role-based access control improve usability and security for staff and management.

15.3 Lessons Learned

The project provided valuable insights and learning opportunities:

- **Team Collaboration:** Clear role distribution and regular communication enhanced productivity and minimized conflicts.
- **System Design:** The MVC architecture and modular approach facilitated easier implementation, testing, and future scalability.
- **Importance of Testing:** Unit, integration, and user acceptance testing ensured a robust and reliable system.
- **Future Planning:** Identifying limitations and potential enhancements early allows for continuous improvement and sustainability of the system.

15.4 Conclusion

Overall, the WMS project demonstrates how software engineering principles can be applied to solve real-world operational challenges in warehouse management. The system provides a reliable, secure, and scalable solution that can evolve with business needs, and sets a foundation for future enhancements such as AI analytics, offline support, and e-commerce integration.