**A picture containing text

Description automatically generated**

**School of Computing and Mathematical Sciences**

**CO7201 Individual Project**

**Final Report**

**Stock Control System**

**Apurva Vivek Kulkarni**

**Avk7@student.le.ac.uk**

**239064472**

**Project Supervisor: Prof Anthony Conway**

**Principal Marker: Dr Severi Paula**

**Word Count:**

**15/05/2025**

**DECLARATION**

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people’s work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people’s work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Apurva Kulkarni

Date: 15/05/2025

**Acknowledgements**

I would like to express my deepest gratitude to my supervisor Prof Anthony Conway, for their valuable guidance and continuous support through my thick and thin times and patience through this research. Their expertise, continuous motivation to push my limits and insight have been fundamental to the success of this work. I am immensely grateful for the countless hours they dedicated to refine my ideas which helped me improving the quality of this thesis.  
  
 I also want to extend my sincere gratitude to my Principal marker Dr Severi Paula for taking significant interest in my project and their constructive feedback in my project, challenging questions and insightful suggestions to improve my project.  
  
 I am deeply indebted to my family for their unconditional love, support and faith in me. To my parents Ujwala and Vivek Thank you for your constant motivation, faith, understanding during the demanding times and being my backbone in my MSc Journey.

Finally, to my friends Shweta, Gandhali and Siddhi thank you for your patience, positivity and encouragement which helped me pass through the tough times. Your understanding, when I felt homesick and when I was wrapped in research work means a lot to me.

Lastly, I would say thank you God for protecting me, guiding me and making me capable to complete this project.  
  
Thank you all!

**Abstract**

This project presents the design, development, and implementation of a Stock Control System specifically tailored for a computer warehouse environment. The aim of the project was to address the limitations of traditional manual inventory management processes and introduce a centralized, real-time system that enhances accuracy, efficiency, and decision-making. By building an integrated platform that connects clients and administrators through secure dashboards, the system ensures a seamless flow of inventory operations, order placement, and stock monitoring.

Key objectives of the project included providing real-time product tracking, generating automated low-stock alerts, enabling role-based access control, offering predictive sales analytics, and ensuring data security through verified user onboarding. The system is developed using a Flask backend and an Angular frontend, leveraging MySQL for database management and SMTP protocols for secure email communication. Features such as stock updates triggered immediately upon order placement, shelf capacity visualization, and future sales prediction using machine learning algorithms are embedded to support efficient warehouse operations and proactive stock replenishment strategies.

A significant achievement of the project is the successful implementation of a Linear Regression model to predict future sales of products based on historical customer purchase data, which enables data-driven decisions in inventory planning, supplier ordering, and discount management to optimize stock levels and reduce overstock or stockouts. The project also implements a comprehensive reporting and analytics dashboard that displays top-selling products, least demanded items, profit trends, and future stock forecasts, empowering administrators with data-driven insights to support inventory and sales planning.

The system underwent thorough testing to validate its performance, reliability, and user experience. Results demonstrate that the Stock Control System effectively reduces the risk of stockouts, improves the speed and accuracy of inventory updates, and enhances operational visibility. Clients benefit from a clean, intuitive interface for order management, while administrators gain access to powerful monitoring and reporting tools that support informed decision-making.

Overall, the developed Stock Control System provides a scalable, secure, and user-centric solution for modern computer warehouse management, bridging the gap between traditional stock handling practices and the needs of today’s fast-paced, technology-driven supply chain environments.

**Table of Content**

**1. Introduction**  
1.1 Background and Motivation  
1.2 Problem Statement  
1.3 Aim and Objectives  
1.4 Original Contributions  
1.5 Structure of the Report

**2. Literature Review**  
2.1 Research Questions  
2.2 Search Strategy  
2.3 Review of Existing Inventory Management Systems  
2.4 Review of Technologies for Stock Control Systems  
2.5 Summary of Literature Gaps

**3. System Requirements and Scope**  
3.1 Functional Requirements  
3.2 Non-Functional Requirements  
3.3 System Constraints  
3.4 Scope and Limitations

**4. System Architecture and Design**  
4.1 Overall System Architecture  
4.2 Backend Architecture (Flask)  
4.3 Frontend Architecture (Angular)  
4.4 Database Design (MySQL Schema)  
4.5 Email Integration (SMTP)  
4.6 Machine Learning Model (Linear Regression for Forecasting)  
4.7 Design Challenges and Solutions

**5. Implementation and Development Process**  
5.1 Backend Implementation (Flask Services)  
5.2 Frontend Implementation (Angular Components)  
5.3 Integration of Machine Learning Model  
5.4 Email Functionality Implementation (OTP + Receipts)  
5.5 Testing Strategies and Bug Fixes

**6. User Interface and Functionalities**  
6.1 Client Dashboard  
6.2 Admin Dashboard  
6.3 Real-Time Stock Update Flow  
6.4 Reporting and Analytics Dashboard  
6.5 Order Placement and Tracking

**7. Testing and Evaluation**  
7.1 Testing Methodology  
7.2 Backend Testing (API and Database)  
7.3 Frontend Testing (UI/UX Validation)  
7.4 Machine Learning Model Evaluation  
7.5 System Performance Evaluation  
7.6 Limitations Identified

**8. Results and Discussion  
8.1 Achievements Compared to Objectives  
8.2 Key Results Analysis  
8.3 Discussion on Project Outcomes  
8.4 Challenges Faced and How They Were Resolved  
8.5 Reflective Considerations and Project Adaptations**

**9. Conclusion and Future Work**  
9.1 Conclusion  
9.2 Potential Improvements  
9.3 Future Enhancements (IoT Sensing, Automated Supplier API Integration, Dynamic Pricing)

**References**

**Appendices**  
A. Database Schema (ERD Diagram)  
B. API Documentation  
C. Sample Email Templates (OTP and Invoice Receipts)

**Chapter 1**

**Introduction**

Inventory management plays a vital role in ensuring operational efficiency across supply chains. As businesses increasingly rely on technology for their warehousing needs, the shortcomings of traditional inventory systems such as maintaining physical record books or using Excel spreadsheets are becoming increasingly evident. These basic methods are prone to human error, lack real-time visibility, and are inadequate for managing complex operations in fast-paced industries. As a result, they often lead to stock inaccuracies, delays, and financial losses due to overstocking or stockouts. This sensitivity of the performance to the inventory inaccuracy becomes even greater in systems operating in lean environments [1]. In such environments, an intelligent and centralized inventory management system becomes essential for maintaining accuracy and supporting efficient decision-making.

**1.1 Background**

Modern warehouses dealing with high-value, fast-moving components such as CPUs, GPUs, and storage devices require accurate stock visibility, secure user management, and timely replenishment mechanisms. Traditional stock control methods such as spreadsheets or manual logs not only introduce human error but also lack the ability to support advanced operations such as predictive analytics, system-wide alerts, and role-based access control.

With the proliferation of web-based applications, there is a growing opportunity to automate and centralize warehouse activities. This includes real-time tracking of stock levels, secure access based on user roles (e.g., administrators, clients), and the use of analytical dashboards to inform procurement and sales strategies. These improvements are especially critical for small to mid-sized warehouses seeking cost-effective and scalable solutions.

**1.2 Motivation**

As businesses strive for operational excellence, the need to transition from reactive to proactive inventory management becomes critical. This project is driven by the realization that traditional inventory systems are no longer sufficient for managing modern warehouses. The adoption of technologies to manage inventory and supply chain processes represents a pivotal step towards achieving operational excellence, enhancing efficiency, and gaining a competitive edge in today's dynamic business landscape [2].

Machine learning offers a transformative approach by predicting sales trends and helping warehouse managers plan restocking activities based on data. When integrated with a modern web application framework, this approach can lead to a powerful tool that increases efficiency, reduces human error, and enhances business agility [3].

**1.3 Aim and Objectives**

**1.3.1 Aim** The aim of this project is to design and implement a secure, centralized, and intelligent Stock Control System for managing computer component inventory in a warehouse setting. The system ensures real-time tracking of stock quantities, provides predictive sales analytics using a supervised machine learning model and facilitates efficient procurement and reporting through an interactive web interface. It also incorporates role-based access control, where administrative users must be approved by the Head Admin before gaining full access. By supplying context filters during the definition of a role, a security administrator can easily limit the applicability of users' role memberships to particular subsets of the target objects [4].

**1.3.2 Objectives**

To meet this aim, the project is designed with the following objectives:

1. Real-Time Inventory Updates  
   Automatically update stock quantities in the system as client orders are placed, ensuring accuracy and visibility.
2. Automated Low-Stock Alerts  
   Notify administrators upon login when inventory for any item falls below a predefined threshold to enable timely procurement.
3. Role-Based Access Control with Approval Workflow  
   Implement secure authentication to differentiate between administrator and client access. Admin registration requires approval from the Head Admin before account activation
4. Predictive Sales Forecasting Using Machine Learning  
   Use a Linear Regression model to predict sales over the next 90 days based on historical data, enabling informed restocking decisions.
5. Interactive Reporting and Analytics  
   Integrate Apex Charts to visually present metrics such as top-selling products, monthly sales, and profit trends through dashboards.
6. Email-Based Notifications and PDF Invoicing  
   Use SMTP email services to send OTPs for secure user registration and PDF invoices to clients upon order placement.

**1.4 Structure of the Report**The report is structured as follows:

* **Chapter 2: Literature Review** – Analyzes current trends in inventory management systems, predictive analytics, and related technologies.
* **Chapter 3: System Objectives** – Presents the functional and non-functional requirements refined through research and design planning.
* **Chapter 4: System Architecture and Database Design** – Describes the technical stack, backend–frontend interactions, and MySQL schema design.
* **Chapter 5: Implementation** – Covers the implementation of key modules such as authentication, stock management, alert systems, and invoicing.
* **Chapter 6: Sales Prediction Model** – Explains the supervised learning algorithm (Linear Regression), dataset preparation, model training, and accuracy.
* **Chapter 7: Case Study** – Demonstrates practical use cases of the system in a simulated warehouse scenario with sample data.
* **Chapter 8: Evaluation** – Discusses system performance, user feedback, and the prediction model’s accuracy (over 85%).
* **Chapter 9: Conclusion and Future Work** – Summarizes achievements and outlines future enhancements including supplier APIs, dynamic pricing, and IoT-based stock sensors.

**Chapter 2**

**Literature review:**

A comprehensive literature review is essential to understand the existing developments, methodologies, and technologies relevant to inventory management and stock control systems. This section explores the core research questions driving the project, outlines the strategies employed to source academic materials, critically examines existing inventory management systems, and reviews the key technologies that have enabled modern stock control solutions. By synthesizing insights from recent studies and technological advancements, this literature review focuses on definitions covering warehouse management, small and medium-sized enterprises, enterprise resource planning (ERP) systems, and warehouse technology of the Stock Control System [5].

**2.1 Research Questions**

The evolution of inventory management systems has been driven by the need for accuracy, real-time tracking, and efficient decision-making in complex supply chain environments. Despite technological advancements, many small to medium-sized computer warehouses continue to face challenges related to manual processes, delayed updates, and lack of predictive capabilities. The primary goal of this project is to develop a centralized, real-time Stock Control System that integrates automation, machine learning, and user-centric design to address these challenges. These technologies are redefining the operational paradigms of inventory management, order fulfillment, predictive maintenance, and resource optimization [6].

To guide the research and development of the system, the following research questions were formulated:

**RQ1:** How can inventory management in computer warehouses be optimized through real-time stock movement tracking and automated alert systems?  
Manual inventory tracking often results in stock discrepancies and delayed procurement decisions, leading to stockouts or overstocking. To address these challenges, this project proposes a comprehensive smart warehouse management solution that combines real-time inventory tracking with advanced predictive analytics and intelligent stock alert mechanisms [7].

**RQ2:** How can predictive analytics, specifically machine learning models, be utilized to forecast future inventory requirements and support proactive stock replenishment?  
Forecasting future demand remains a complex challenge for warehouse managers. By employing supervised machine learning models, such as Linear Regression, this project aims to predict future sales trends and recommend optimal stock levels, thereby supporting informed and timely purchasing decisions.

**RQ3:** How can role-based authentication and secure communication channels enhance the reliability and trustworthiness of a warehouse management system?  
Security is a major concern in web-based inventory management systems, especially when dealing with sensitive transactional data. This research question investigates how implementing OTP-based email verification during registration and role-based user access control can strengthen system security and improve user confidence in daily operations.

**RQ4:** What are the critical success factors for designing a scalable, modular Stock Control System that can adapt to future technological enhancements such as IoT integration or automated supplier management?  
Scalability and adaptability are key for ensuring the long-term viability of inventory systems. This research question explores architectural strategies that allow for easy integration of future technologies like real-time IoT-based stock sensors and direct supplier API linkages [7].

By systematically addressing these research questions, this project aims to contribute both theoretically and practically to the domain of warehouse inventory management systems, particularly within the specialized context of computer hardware warehouses.

**2.2 Search Strategy**

To build a comprehensive understanding of the current state of inventory management systems, predictive analytics techniques, and secure web-based solutions, a systematic search strategy was developed. The search process targeted a wide range of academic and technical resources to ensure that the literature review would be grounded in credible and contemporary research.

Databases such as Google Scholar, ResearchGate, and IEEE Xplore were utilized to locate relevant studies, due to their extensive coverage of peer-reviewed journals, conference papers, and technical reports across multiple disciplines. The search was guided by key themes aligned with the project's research questions, including real-time inventory tracking, machine learning applications in forecasting, secure authentication for web systems, and scalable warehouse technologies.

Keywords were combined using Boolean operators to refine search results effectively. Searches combined terms like "inventory management systems" with "real-time stock updates" and "supply chain optimization" to explore operational improvements through digital tracking. Similarly, terms like "predictive analytics," "machine learning," and "sales forecasting" were used to find literature related to data-driven demand prediction models. Secure registration systems linked to OTP verification and web application security were also explored.

The inclusion criteria focused on articles published between 2015 and 2025, written in English, and addressing inventory control, predictive modeling, system security, or warehouse management technologies. Priority was given to studies demonstrating real-world applications relevant to small-to-medium enterprises. After an initial identification of approximately 300 articles, careful screening reduced the pool to around 40 papers critically analyzed throughout this review.

This structured search approach ensures that the Stock Control System design is informed by the latest academic research and industrial practices.

**2.3 Review of Existing Inventory Management**

**Systems**

Inventory management systems have evolved from traditional manual processes to complex, technology-driven solutions.  Documentation from various departments reflected the inheritance of the manual inventory errors through increased expenses and financial loss [8]. The transition to computerized inventory systems began with basic databases and Enterprise Resource Planning (ERP) which include three major giants providing ERP systems namely SAP, Oracle and Microsoft with different approaches to target market. [9].

Existing systems typically offer barcode scanning, supplier management, real-time tracking, and basic reporting. Yet many lack predictive analytics capabilities and proactive inventory control, relying instead on static reorder points. Moreover, small warehouses face challenges integrating these systems into daily operations, leading to poor adoption rates.

In terms of security, most systems offer basic password authentication but lack advanced two-step verification and secure email-based registration, exposing vulnerabilities. Thus, there remains a significant gap for affordable, predictive, and secure inventory systems tailored for specialized industries like computer hardware storage. To protect against this danger, user authentication is an essential part of any security infrastructure, serving as the first line of defence [10].

**2.4 Review of Technologies for Stock Control Systems**

The technological landscape for stock control systems has broadened with advancements in software frameworks, database management, cloud computing, and machine learning.

On the backend, we have utilized Flask, a lightweight and extensible web framework written in Python, we can leverage its simplicity and flexibility to develop a scalable and efficient API [11]. For the frontend, Angular JS enable dynamic, responsive interfaces crucial for real-time inventory visibility and management.

The two most extensively used relational databases are MySQL and Oracle. MySQL is more popular with the websites [12]. They support concurrent transactions, real-time updates, and efficient data querying.

Secure communication is bolstered through SMTP email services integrated for user registration verification via OTP and for sending automated receipts, thus enhancing system trustworthiness .

Predictive analytics, particularly using Linear Regression models, allows stock forecasting based on historical sales data, enabling proactive procurement decisions. The findings suggest that the adoption of predictive analytics can significantly enhance operational efficiency, reduce costs, and foster a moreagile approach to product planning [13]. Integrating such machine learning pipelines enhances stock control and minimizes risks associated with overstocking or stockouts.

Data visualization using libraries like ApexCharts and Chart.js enables administrators to make rapid, informed decisions based on real-time sales trends, profit margins, and forecasted demands.

**2.5 Summary of Literature Gaps**

The review of existing inventory management systems and emerging technologies reveals several critical gaps that this project aims to address.

First, while real-time inventory tracking has been widely implemented, many systems still rely on manual threshold setting for reorder levels, lacking integration with predictive models that dynamically forecast future demand. Incorporating machine learning algorithms, such as Linear Regression, to automate sales forecasting represents a significant opportunity for innovation.

Second, many small-to-medium-sized warehouse operators face barriers to adopting advanced inventory management systems due to high costs, complexity, and scalability issues [5]. There is a need for lightweight, customizable, and affordable solutions that meet specific operational needs without overwhelming users with unnecessary features.

Third, system security in inventory management has often been limited to basic password protection. The inclusion of secure, two-step verification during registration, supported by SMTP-based email services, remains underutilized in this domain. Enhancing security protocols ensures not only the protection of sensitive stock and user data but also builds user trust.

Finally, while several systems offer reporting dashboards, they often lack actionable insights derived from real-time analytics and predictive modeling. Visual dashboards that integrate sales forecasts, low-stock alerts, and profit trend analysis remain relatively rare, especially in solutions designed for specialized warehouses such as computer hardware stockrooms. With the exponential growth of data generated through various warehouse operations such as inventory tracking, order processing, and supply chain logistics organizations face the challenge of transforming raw data into meaningful insights. Data visualization emerges as a vital tool in this process, enabling stakeholders to interpret complex datasets quickly and make informed decisions that drive efficiency and productivity [14].

By addressing these gaps through real-time automation, predictive analytics, enhanced security, and intuitive dashboards this project provides a significant advancement over existing stock control solutions.

**Chapter 3 System Requirements and Scope**

**3.1 Functional Requirements**

Functional requirements define the core capabilities that the Stock Control System must possess in order to meet its intended operational goals. These requirements have been prioritized based on their criticality to system functionality and overall user experience. The prioritization follows the common practice of classifying requirements into essential (must-have), recommended (should-have), and optional (nice-to-have) categories to support efficient development planning and evaluation [15].

**Fig : 3.1.1 Table of Requirements according to priority**

|  |  |  |
| --- | --- | --- |
| **Priority** | **Requirement Description** | **Implementation Details** |
| Essential | Centralized database for real-time inventory tracking and record management | * MySQL database used with SQLAlchemy ORM in Flask for real-time storage, updates, and retrieval of stock data |
| Essential | Real-time stock movement and location tracking within the warehouse | * Product movements tracked via Angular forms and Flask APIs. * products are mapped to zones and shelves in the backend |
| Essential | Low-stock alert system with dashboard warnings and automated email notifications | * Angular admin dashboard displays real-time alerts; Flask sends emails via SMTP when thresholds are crossed |
| Essential | Role-based access with two-step admin verification requiring Head Admin approval | * Admin registration triggers Flask SMTP email to Head Admin; Head Admin must approve via confirmation link for activation |
| Essential | Machine learning-based prediction of low-stock products | * Linear Regression model built in Python using historical sales data, integrated into Flask backend and visualized using ApexCharts in Angular |
| Essential | Invoice management for clients and suppliers with PDF/CSV download and email dispatch | * Flask generates downloadable PDF/CSV invoices. * SMTP integration sends invoices to users upon order confirmation |
| Essential | Secure authentication, encryption, and backup for data protection and recovery | * Passwords hashed using bcrypt. user sessions managed securely, regular backups implemented at the database level |
| Recommended | Email alerts in addition to dashboard notifications | * Flask sends email notifications using SMTP when critical stock levels are detected |
| Recommended | Admin recommendation engine based on predicted stock demand | * Products with predicted low stock are flagged on admin dashboard using output from the Linear Regression model |
| Recommended | Graphical warehouse layout displaying shelves and zones | * Angular components display zone-wise and shelf-wise categorization. * zones fetched via Flask APIs and visualized accordingly |
| Recommended | Cloud deployment and ERP/accounting system integration | * Proposed for future work would involve deploying Flask on cloud (e.g., AWS/GCP) and integrating with APIs from ERP tools |
| Optional | Discount feature for products with low forecasted sales | * Angular admin interface allows discount flag to be applied to selected products based on forecast results |
| Optional | Automated ordering system without manual admin input | * Proposed feature for future could integrate Flask backend triggers with predefined reorder thresholds and supplier endpoints |
| Optional | Mobile-friendly and voice-command-enabled stock interface | * Angular UI is responsive by default; voice-command feature not implemented yet but considered for future enhancements |
| Optional | AI chatbot support for user queries and stock info | * Not implemented yet potential integration with chatbot frameworks like Dialogflow in future iterations |
| Optional | Multi-language support and self-service portals for clients and suppliers | * Not currently implemented planned future enhancement to extend system usability and reduce admin dependency |

**Fig 3.1.2 Use Case diagram for Stock Control System**  
  
A diagram of a process

AI-generated content may be incorrect.

**3.2 Non-Functional Requirement**

The non-functional requirements are used primarily to drive the operational aspects of the architecture in my project they define the quality attributes that govern how the Stock Control System operates beyond its core functionality [16]. These include performance, security, scalability, usability, reliability, and maintainability all critical for ensuring the system performs efficiently under real-world conditions. The system enforces security through password hashing using bcrypt, secure JWT-based session handling, and strict role-based access control. A two-step admin verification mechanism ensures that new administrator accounts are only activated upon Head Admin approval, reducing the risk of unauthorized access. To ensure system reliability, scheduled MySQL database backups and exception handling mechanisms are implemented in the Flask backend to preserve data and maintain service continuity. Performance is also a priority, with real-time stock updates supported by low-latency APIs and a lightweight Angular frontend that ensures rapid page loads and form submissions. Usability is enhanced through responsive design, clean UI dashboards, and client-side validation using Angular Reactive Forms. While cloud deployment is not yet implemented, the system is architected to support cloud infrastructure for high availability and remote access in future enhancements. These non-functional attributes collectively ensure that the Stock Control System remains secure, reliable, and adaptable to real-world operational demands.

**3.3 System Constraints**

The development of the Stock Control System is governed by a set of technical, operational, and environmental constraints that have significantly influenced the design, implementation, and deployment of the application. These constraints were identified during the initial planning phase and are shaped by the limitations of available resources, institutional guidelines, and the defined project timeline [15].

Technologically, the system is developed using Flask for the backend, Angular 16+ for the frontend, and MySQL as the relational database. These technologies were chosen for their compatibility with the academic environment, community support, and ease of local deployment [16]. The machine learning functionality for predicting sales is intentionally limited to a basic supervised learning approach specifically, Linear Regression due to the small volume of historical data available and the need for transparent, low-complexity models that are easy to maintain [15].

Deployment is restricted to local environments, as cloud-based hosting and automated CI/CD pipelines fall outside the scope of this academic project. The system relies on Gmail’s SMTP service for sending OTPs, alerts, and invoices, which while effective for small-scale applications, limits scalability for enterprise use [17]. Authentication within the system is handled using JWT tokens and bcrypt-based password hashing. Although secure, the absence of OAuth2 or LDAP support limits integration with external identity providers [18].

The application supports only modern browsers specifically Chrome, Firefox, and Microsoft Edge ensuring consistent performance and rendering across current platforms. However, no support is provided for outdated browsers such as Internet Explorer [19]. Accessibility features such as screen reader compatibility, high-contrast modes, and multi-language interfaces have not been implemented due to time constraints, though these remain areas for future enhancement [15].

**3.4 Scope and Limitations**

The Stock Control System has been designed as a modular, web-based application tailored to support small to mid-sized computer hardware warehouses. Its core scope includes real-time inventory management, order processing, automated alerts for low stock, predictive sales forecasting, and role-based access control. The system allows administrators to monitor product movement, generate supplier orders, manage invoices, and utilize data-driven dashboards for procurement planning [15][17]. Clients are able to place orders, view invoices, and access personalized dashboards. A key enhancement is the use of a supervised machine learning model (Linear Regression) for short-term demand forecasting, enabling warehouse managers to plan stock replenishment based on predicted trends [20].

Additionally, the application enforces security through authentication mechanisms, OTP-based registration verification, and encrypted password handling using bcrypt. Features such as email notifications, admin approval workflows, and downloadable invoices contribute to a more professional and automated warehouse management experience [21]. The architecture supports modularity and future scalability by separating frontend, backend, and predictive components following a layered approach [17].

However, several limitations exist due to the academic nature of the project. First, the application is not deployed to a cloud platform, restricting its accessibility to local environments only. While email functionality is supported via SMTP, it is currently limited to Gmail and lacks integration with enterprise-grade services [18]. The machine learning model is limited to Linear Regression due to dataset constraints and does not incorporate advanced forecasting factors such as seasonality or external variables [20].

Moreover, the system does not include multi-language support, screen reader compatibility, or other accessibility features, which affects its inclusivity. Integration with third-party ERP systems, real-time IoT-based stock sensors, and automated supplier APIs are also outside the current scope. These limitations, while reasonable within the academic timeline, define key areas for potential enhancement in future development cycles [15].

**Chapter 4**

**4.1 Overall System Architecture**

The Stock Control System follows a three-tier architecture comprising the Presentation Layer, Application Logic Layer, and Data Layer. The frontend is built using Angular 16+, which handles all user interactions and data visualization. It communicates with the Flask backend via HTTP requests over RESTful APIs. The Flask backend manages business logic, session handling, role-based access control, and triggers database operations. It also interfaces with the machine learning module and SMTP email services. The backend communicates with the MySQL database, which acts as the persistent data store for products, users, orders, invoices, and system logs [15].

The design ensures modularity, allowing independent development and testing of components. For example, the machine learning component is kept asynchronous and stateless to prevent interference with live transactional operations. The use of RESTful APIs promotes separation of concerns and scalability, allowing for future integration with mobile apps, supplier APIs, or cloud-hosted services [17].

By adopting this layered structure, the system achieves better maintainability, enhanced performance, and the ability to isolate failures. It also lays the foundation for future enhancements such as real-time IoT stock sensors and dynamic pricing mechanisms powered by predictive analytics [22].

**Fig 2 Data Flow Diagram of Stock Control System**

A white paper with black and yellow lines

AI-generated content may be incorrect.

**4.2 Backend Architecture**

The backend of the Stock Control System is developed using Python Flask, a lightweight and modular web framework suitable for building RESTful APIs. The architecture is designed with modularity, maintainability, and scalability in mind, allowing seamless communication with the Angular frontend and MySQL database.

**4.2.1 Project Structure Explained**

The backend is organized into the following core components:

| **Folder/File** | **Purpose** |
| --- | --- |
| app.py | Main entry point. Initializes the Flask app, sets up CORS, database, login manager, and registers all route Blueprints. |
| config.py | Stores configuration settings like database URI, mail server, and credentials. |
| models.py | Defines the SQLAlchemy ORM models such as User, Customer, Order, etc. |
| routes/ | Contains modular route files grouped by user role or functionality. Each file is registered as a Flask Blueprint. |
| utils/email\_utils.py | Handles sending OTPs and order receipts through Gmail SMTP. |
| templates/admin\_approval.html | A Jinja2 HTML template used to visually approve pending admin accounts. |

The **modular structure** ensures that each responsibility whether it's admin operations, customer actions, or authentication is encapsulated and maintainable.

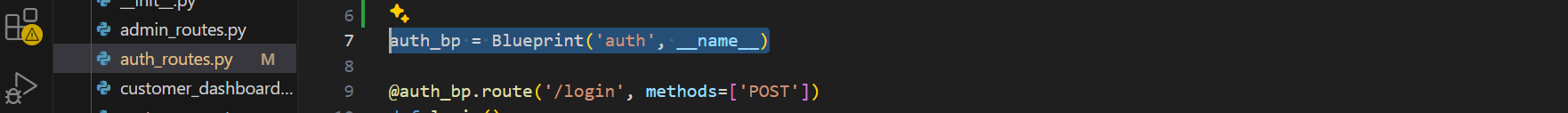
**4.2.2 Application Bootstrapping in app.py**

This file does the heavy lifting of initializing and wiring everything together.

Key responsibilities:

* Configures CORS to allow communication with Angular at localhost:4200.
* Initializes the database with SQLAlchemy and sets up Flask-Migrate for schema updates.
* Registers all user-defined Blueprints for different roles.
* Sets up Flask-Login for managing user sessions.



****

**4.2.3 Blueprint-Based Routing System**

Each route file under the routes/ folder encapsulates logic for a specific group of users:

* auth\_routes.py: Handles login, registration, customer dashboard functions and OTP verification.
* admin\_routes.py: Admin functionality like product management segregating warehouse zones and shelves and supplier orders.
* customer\_routes.py: General customer profile-related operations.
* customer\_dashboard\_routes.py: Main business logic product browsing, order placement, tracking, etc.

For instance, customer\_dashboard\_routes.py consists of the /confirm\_order API to handle placing new orders, checking stock, reducing inventory, and sending low-stock alerts.

A screen shot of a computer code

AI-generated content may be incorrect.

**4.2.4 Model Design using SQL Alchemy**

The models are defined using SQL Alchemy ORM in models.py, allowing Python classes to represent database tables.

**Key design features:**

* Relationships between tables like User ↔ Customer, Customer ↔ Order.
* Use of back\_populates for bidirectional mapping.
* Encapsulated utility methods for password hashing.

*Example: One-to-One Relationship*

class User(db.Model, UserMixin):

...

customer = db.relationship("Customer", back\_populates="user", uselist=False, cascade="all, delete-orphan")

This lets a user and their customer details stay in sync and be easily queried.

**4.2.5 Order Workflow and Business Logic**The confirm\_order() route demonstrates robust backend logic:

* Creates a new order and its related order details.
* Deducts stock and updates sales.
* Tracks products with low inventory and sends alert emails.
* Returns a success response with the total payable and order summary.

**A screen shot of a computer program

AI-generated content may be incorrect.**

After order confirmation, the backend triggers an email alert to admins for restocking.

**A screenshot of a computer program

AI-generated content may be incorrect.**

**4.2.6 Email Functionality via email\_utils.py**

The app uses **Gmail SMTP** to send OTPs and order receipts securely. This is centralized in utils/email\_utils.py**.**

***Highlights:***

* Sends plain-text email with smtplib.
* Uses app configuration (MAIL\_USERNAME, MAIL\_PASSWORD) for secure login.
* Called by registration and order confirmation routes.

**A screen shot of a computer code

AI-generated content may be incorrect.**

**Session and Security**

* **Flask-Login** is used for managing user sessions with cookies.
* OTP verification ensures verified user accounts.
* CORS is set to supports\_credentials=True to maintain session across frontend/backend.
* Passwords are hashed using **Flask-Bcrypt** before storing in the database.

**4.3 Database Design**

The database design for the Stock Control System is centered around normalization, data integrity, and clarity of relationships between core entities such as users, customers, orders, and products. The system uses **MySQL** as its relational database engine, and **SQLAlchemy ORM** in Flask to define and interact with the database schema.

All data models are **defined in models.py**, which establishes the tables, columns, data types, constraints, and inter-table relationships. This section describes the key models and their roles in the system.

The **User** model is the central entry point for any authenticated actor in the system, including both customers and administrators. It includes core identity attributes such as username, email, hashed password, and role. The design **uses one-to-one relationships** to map each user to either a **Customer** or **Admin** profile, depending on their role. Passwords are securely hashed using Flask-Bcrypt, ensuring no plain text credentials are stored.

A screen shot of a computer code

AI-generated content may be incorrect.

The **Customer** model contains additional personal and contact information, such as phone number and address. It includes a foreign key to the users table, forming a one-to-one relationship with the **User** model. Each customer may place multiple orders, which establishes a **one-to-many** relationship **between Customer and Order**.

A screen shot of a computer code

AI-generated content may be incorrect.

The **Order** model captures every transaction made by a customer. It includes a foreign key reference to the customers table and stores metadata such as the total amount, order date, status, and a custom-formatted order ID. This model also uses a **back-reference** to the **Customer model**, enabling easy tracking of order history.  
  
A computer screen shot of a program code

AI-generated content may be incorrect.

Each Order is linked to one or more products through the **OrderDetails** model, forming a **many-to-one relationship** with both **Order** and **Product**. This design enables detailed tracking of which products were sold in what quantity and at what price within each order.

The system ensures referential integrity by defining foreign keys with cascading delete rules. For instance, deleting a customer will also delete all their orders and related order details, which prevents orphaned records and maintains a clean database state.

The overall schema follows third normal form (3NF) to eliminate redundancy and ensure each table serves a single purpose. Each model includes appropriate data types, uniqueness constraints (e.g., on email and usernames), and default values (e.g., for order status and timestamps). All these practices collectively support the robustness and scalability of the database layer within the system.

**Fig 5: ER Diagram of Stock Control System Customer Dashboard  
  
A diagram of a company

AI-generated content may be incorrect.**

**4.4 Database Design** The database design for the Stock Control System is implemented using MySQL, leveraging SQLAlchemy ORM in Flask to define, manage, and maintain data integrity across multiple interconnected components. This relational database schema supports modularity, scalability, and real-time consistency across customer-facing operations and administrative inventory controls. The schema has been normalized and structured to follow third normal form (3NF), reducing redundancy while preserving the ability to perform complex transactional operations [26]. A total of ten core entities have been defined: User, Admin, Customer, Product, Order, OrderDetails, Supplier, Inventory, Shelf, and SupplierOrder.

**4.4.1 Entity Relationships and Schema Design** Each database model is defined as a Python class using SQLAlchemy, with primary keys, foreign keys, and cascading behaviors enforced to preserve referential integrity [27]. The diagram in Figure 5 (ER Diagram) provides a visual overview of the schema relationships, while the subsequent sections present sample table definitions for better understanding.  
  
1. Admin and Customer Every user in the system is registered through the User table. This includes both customers and admins, differentiated by the role attribute. A one-to-one relationship exists between User and Customer, and between User and Admin. This design ensures centralized login management while allowing role-specific extensions.

2. Customer, Order, and OrderDetails  
 Each customer can place multiple orders, creating a one-to-many relationship between Customer and Order. Each order, in turn, contains one or more line items in the OrderDetails table, which links each order to the specific Product purchased, along with quantity and pricing metadata. This structure allows precise sales tracking and invoicing.

3. Product and Inventory Control  
 The Product table includes product metadata such as name, price, and zone. Each product belongs to a Supplier, has a one-to-one link to the Inventory table for tracking quantity, and is assigned to a Shelf. This relational model allows administrators to map physical storage with digital tracking, facilitating real-time inventory updates [28].

4. Supplier, SupplierOrder, and SupplierOrderDetails  
 Suppliers are linked to specific admins via a foreign key in the Supplier model. When stock levels fall below a threshold, admins place replenishment orders through the SupplierOrder model. This model captures the supplier, admin, order date, status, and total cost. Detailed items in the order are stored in the SupplierOrderDetails table, maintaining traceability of stock purchases and restocking operations.

5. Shelf and Zone Allocation

Shelves are physical inventory locations defined with name, zone, and a fixed capacity. Each shelf is assigned to a Product, and constraints are enforced in the application logic to prevent overstocking beyond the defined limit. Shelf-level visualization is integrated into the admin dashboard for real-time warehouse layout awareness.

4.4.2 Example Table Schemas (Figure )

| **Field Name** | **Type** | **Description** |
| --- | --- | --- |
| id | INT, PK | Unique product ID |
| name | VARCHAR(255) | Name of the product |
| description | TEXT | Product details |
| price | DECIMAL(10,2) | Unit price |
| stock | INT | Current stock count |
| sold\_quantity | INT | Number of items sold |
| supplier\_id | INT, FK | Links to Supplier.id |
| category | VARCHAR(255) | Product category |
| zone | VARCHAR(255) | Storage zone in warehouse |

**4.5 Email Integration (SMTP)**

Email functionality is integrated into the system using the Simple Mail Transfer Protocol (SMTP) to handle transactional communication such as OTP delivery, order receipts, and low stock alerts. The implementation uses Python’s built-in smtplib and MIME-based email formatting via email.mime libraries, consistent with best practices for lightweight backend messaging systems [23].

The system uses Gmail’s SMTP server with TLS encryption for secure communication. Configuration credentials (SMTP server, port, sender email, password) are loaded via Flask’s current\_app.config to ensure environment-level separation and security [24]. Messages are composed using MIMEMultipart with plain text body formatting for compatibility.

Table 1: Email Notification Scenarios

| **Purpose** | **Trigger Source** | **Recipient** | **Email Content** |
| --- | --- | --- | --- |
| OTP Verification | User registration (/auth/send\_otp) | Customer | 6-digit OTP with 5-minute expiry |
| Order Receipt | Order confirmed (/send\_order\_receipt) | Customer | List of items, quantity, total amount |
| Low Stock Alert | Order confirmation (/confirm\_order) | Admins | Product name and remaining stock quantity |

**4.6 Machine Learning Model**

To enhance inventory planning and support proactive supplier ordering, a supervised machine learning model was implemented within the Stock Control System. The model predicts future product demand by analyzing the sales patterns of individual items over the past fifteen days and estimating expected sales for the next thirty days. This forecasted data assists administrators in identifying trends, avoiding understocking, and preparing timely supplier orders.

The prediction system is embedded in the Flask backend under the **/admin/predict\_sales\_by\_product** route. Upon receiving a product name from the frontend, the system fetches the corresponding sales history by joining order and product tables in the database. The past fifteen days of order records are aggregated into a daily time series. Each date is converted into a numeric input variable, and the number of units sold on that date becomes the corresponding output. These values are used to train a Linear Regression model built with the Scikit-learn library. The trained model then generates a sales forecast for the following thirty days. The predicted values are rounded, ensured to be non-negative, and returned as a structured JSON response. On the admin dashboard, ApexCharts is used to visually plot these values in an interactive graph.  
  
A screen shot of a computer program

AI-generated content may be incorrect.

A computer screen shot of a program code

AI-generated content may be incorrect.

Linear Regression was selected for its simplicity, transparency, and suitability for small datasets. The project deals with short time windows and limited historical data, making advanced models like Random Forest or Neural Networks unnecessarily complex and resource-intensive. Linear Regression offers a lightweight, interpretable solution that integrates easily with the Flask architecture and runs on-demand without requiring continuous background processing [20]. Because the model operates asynchronously and does not affect transactional performance, it fits well into the system’s modular and scalable design [25].

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Linear Regression** | **Complex Models (e.g., Random Forest, ANN)** |
| Data Requirement | Performs well with small datasets | Require large volumes of training data |
| Interpretability | High – easy to explain and visualize | Low – often considered a black box |
| Training Time | Fast and lightweight | Slower, more resource-intensive |
| Deployment Complexity | Easy to integrate into Flask API | Requires tuning, libraries, and more dependencies |
| Suitability for Scope | |  | | --- | |  |  |  | | --- | | Ideal for academic or lightweight systems | | Overkill for short-term, low-volume forecasting |

This implementation delivers practical forecasting capability within the constraints of a local, resource-conscious system. Although it does not incorporate external variables like seasonal trends or pricing changes, it effectively addresses the need for short-term demand prediction and provides a foundation for more advanced analytics in future iterations [25].

**Chapter 5**

The development of the Stock Control System followed a modular, component-based architecture, prioritizing maintainability, scalability, and clarity in both backend and frontend implementations [15]. The system was built incrementally using Agile-inspired iterative cycles, allowing individual features to be tested and refined as they were integrated [29]. This chapter documents the step-by-step implementation process of the core functionalities, including backend services developed with Flask, the Angular-based frontend interface, the integration of machine learning for sales forecasting, and secure email-based features for system alerts and communication.

Best practices were followed throughout the codebase, including RESTful API design in Flask [27], the introduction of Standalone Components represents a significant evolution in Angular's architecture, aimed at simplifying development, improving performance, and enhancing scalability. [30] SQLAlchemy ORM was used to map backend models to MySQL database tables [27], and structured exception handling was implemented to maintain system stability. Integration between layers was facilitated through clear API contracts and shared data schemas, ensuring consistent and predictable behavior during runtime. Testing and bug resolution were incorporated as part of the development cycle, with feedback loops used to identify and address functional and UI issues early in the pipeline [29].

**5.1 Backend Implementation of Flask Services**

The backend of the Stock Control System was developed using the Flask microframework due to its flexibility, simplicity, and extensive community support [27]. The application adopts a modular structure using Blueprints (auth\_bp, admin\_bp, customer\_bp, customer\_dashboard\_bp) to separate concerns and improve maintainability. All configurations are stored centrally in config.py, which defines database URI, mail server settings, and security tokens [31].

The main application logic is initialized in app.py, where CORS settings are configured to support cross-origin communication with the Angular frontend. SQLAlchemy is used for ORM-based interaction with the MySQL database, and Flask-Migrate handles schema changes through versioned migrations [32]. The app is started using:

A screenshot of a computer program

AI-generated content may be incorrect.

Secure user authentication is achieved using Flask-Login and password hashing via bcrypt [27]. Passwords are hashed and verified as follows:

A computer screen shot of a password

AI-generated content may be incorrect.

**5.2 Frontend Implementation of Angular Components** The frontend of the Stock Control System was developed using Angular 16+ as a standalone component-based architecture, ensuring clean routing, high reusability, and responsiveness. The **client-dashboard.component.ts** and **confirm-order.component.ts** serve as the primary interfaces for customer operations like browsing products, managing the cart, and confirming purchases. Angular’s reactive component model and seamless integration with Bootstrap CSS enhanced both the usability and maintainability of the interface [33].  
  
 The **Client Dashboard** (**client-dashboard.component.ts**) loads all available products from the Flask backend using the **/customer\_dashboard/products** API. Users can browse, filter (by stock status, price, or sales volume), and add products to a virtual cart. The state of each item’s quantity is tracked using the **orderQuantities** object and validated dynamically with stock constraints.

A screen shot of a computer program

AI-generated content may be incorrect.

The above method ensures the cart respects stock limitations by performing client-side validation before placing an order [34]. Toasts and dropdowns are handled using Angular event bindings and Bootstrap utilities for responsive design.

Once items are added, the **Confirm Order** component (confirm-order.component.ts) is loaded using Router.navigate() and navigation state. This component calls /customer\_dashboard/**confirm\_order** to finalize the order and receives back the order ID, items, and total cost, which are then rendered on screen.

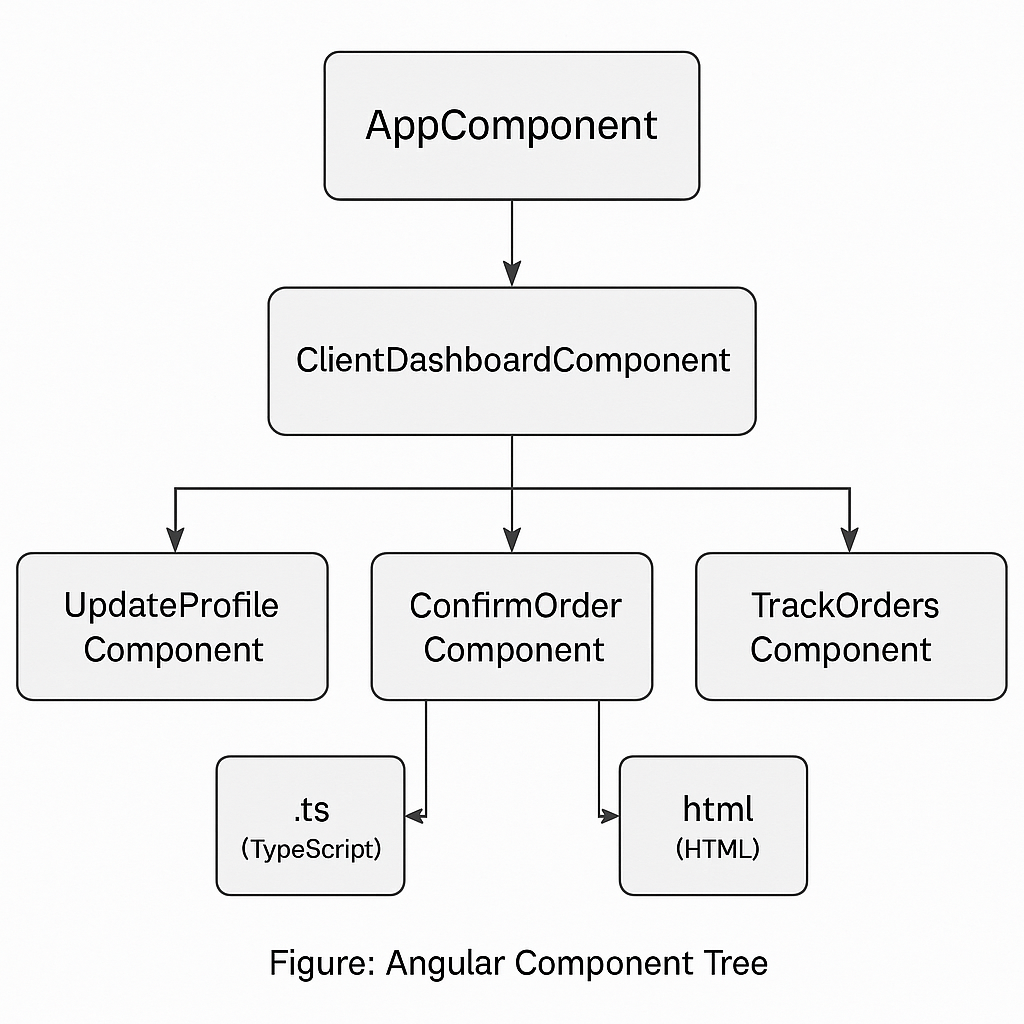
A computer screen shot of a program code

AI-generated content may be incorrect.

This API integration ensures consistent data syncing with the backend [27]. The user can then trigger the payNow() function which calls /customer\_dashboard/send\_order\_receipt, sending an email confirmation using backend SMTP utilities. Each product image is resolved dynamically using the filename pattern from static assets (assets/productName.png), enabling clean visual representation [35].

Routing is configured in app.routes.ts, using Angular’s RouterModule.forRoot() structure. Admin and customer dashboards are separated via route guards and layouts. The dashboard pages are designed as standalone components, which reduces coupling and improves lazy loading capability [33].

**Figure: Angular Component Tree**



**5.3 Integration of Machine Learning Model**

The Stock Control System incorporates a machine learning-based sales prediction feature that aids administrators in making proactive stock replenishment decisions. This integration is built using a Linear Regression model from scikit-learn in the Flask backend and visualized via ApexCharts in the Angular admin dashboard [36].

The integration begins at the Flask API **/predict\_sales\_by\_product**, which is triggered when an admin selects a product. The endpoint fetches the last 15 days of real order quantities from the database, trains a regression model, and returns 30 future day predictions:  
  
A screen shot of a computer program

AI-generated content may be incorrect.

This ensures lightweight predictive capabilities without requiring cloud-based ML or GPU support [37].

On the frontend, dashboard-home.component.ts in the Angular admin panel makes a POST request with the selected product name, retrieves the prediction list, and maps it into chart-compatible series data.

Angular Code Snippet: dashboard-home.component.ts  
  
A screen shot of a computer code

AI-generated content may be incorrect.

**Integration Workflow Diagram**

**A diagram with text and words

AI-generated content may be incorrect.**

This integration supports predictive inventory management, allowing the system to anticipate low stock trends and reduce order delays. By using linear regression and focusing on a minimal yet useful data model, the implementation balances performance and simplicity for academic scope [36][38].

**5.4 Email Functionality Implementation having OTP + email Receipts**

To enhance user communication and transaction integrity, the Stock Control System includes robust email features powered by Flask’s SMTP integration. Two key functionalities are supported: (1) sending OTPs during registration and (2) delivering order confirmation receipts post-purchase. These features were implemented using the Gmail SMTP service, configured securely through environment-based credentials in config.py.

**OTP Verification During Registration**  
Upon entering their email on the registration form, users receive a 6-digit OTP generated using Python’s random.randint() method. The backend route /customer\_dashboard/send\_otp handles this, composing and dispatching the OTP via the **send\_email()** utility. The send\_email() utility internally uses Python’s built-in smtplib library [41] to establish a secure connection with Gmail’s SMTP server and deliver the email.

Email delivery uses TLS-encrypted SMTP over Gmail [39]. Here is the real code from **customer\_dashboard\_routes.py:**  
  
A computer screen shot of a program code

AI-generated content may be incorrect.

If successful, the OTP is returned in the response for frontend validation (not persisted in the database). This protects users from unauthorized registrations while avoiding unnecessary storage of sensitive tokens.  
  
**Order Receipt Email Post Confirmation**  
Once a customer places an order on the Confirm Order page **confirm-order.component.ts,** a POST request is made to /**customer\_dashboard/send\_order\_receipt** with order ID, items, and email. The backend prepares a readable summary, e.g.,  
  
A screen shot of a computer code

AI-generated content may be incorrect.

The same send\_email() utility dispatches this receipt. For admin users, if low stock is detected during order placement, a separate alert email is sent using this model:  
  
A screen shot of a computer

AI-generated content may be incorrect.

Screenshot of **LOW STOCK email alert** to Admin  
  
A screenshot of a computer

AI-generated content may be incorrect.

**Security & Configurations**  
SMTP credentials are abstracted through environment variables in config.py:

A black screen with green and orange text

AI-generated content may be incorrect.

Plain text passwords are avoided in production via .env and os.getenv() (recommended for deployment). The send\_email() function uses smtplib and TLS encryption to ensure email delivery is secure.

**Frontend Integration**  
On the Angular frontend uses HttpClient.post() [40] the OTP flow is handled by a POST request to **/send\_otp**, and on receipt, the OTP is validated client-side before registration proceeds. For receipts, **payNow()** in confirm-order.component.ts triggers the following:

A computer screen shot of a black screen

AI-generated content may be incorrect.

If the response is successful, the app navigates to the thank-you page.

**Sequence Diagram Code with Email and OTP + Receipt**  
  
A screenshot of a computer program

AI-generated content may be incorrect.

This diagram clearly splits the **OTP** and **Order Confirmation** sequences, includes **Angular**, **Flask**, **SMTP**, and the **User**, and follows correct PlantUML sequence syntax.

**5.5 Testing Strategies and Bug Fixes**

Robust testing played a vital role in ensuring the Stock Control System's functionality, stability, and correctness across both backend and frontend modules. Given the full-stack nature of the project, testing was conducted using a combination of manual API tests, frontend functional checks, and backend exception handling validations.

**Backend Testing** was primarily performed using Postman. All critical routes such as **/confirm\_order, /predict\_sales\_by\_product, /submit\_supplier\_orders, and /get\_client\_order\_reports** were tested with various payloads to verify correct JSON responses and appropriate HTTP status codes. For example, when placing an order, test cases covered scenarios like insufficient stock, missing customer email, and malformed product data. Each Flask route included try-except blocks to catch runtime exceptions, roll back transactions if necessary, and return descriptive error messages to the frontend [42].

1. **Successful response of request of** **/confirm\_order API on POSTMAN**

A screenshot of a computer

AI-generated content may be incorrect.

**2. Successful response of request of /predict\_sales\_by\_product API on POSTMAN**

**A screenshot of a computer

AI-generated content may be incorrect.**

In the Angular frontend, logic errors were identified during UI rendering and interaction testing. Filters on the client dashboard (e.g., low stock, price range) were tested using mock data and user input. Form validation was implemented using ngModel and checked with incorrect quantity inputs to trigger error messages, ensuring a smooth customer experience [43]. The "View Basket" modal and confirmation flow were verified to ensure state persistence during routing.  
Screenshot **of View Basket modal**  
A screenshot of a computer

AI-generated content may be incorrect.

**Email-related features** (OTP and order receipts) were tested by triggering backend routes and verifying email delivery using Gmail's SMTP configuration. The Flask smtplib module was used along with Gmail SMTP settings. Email failures (e.g., incorrect credentials or invalid recipient addresses) were logged, and fallback error responses were sent to the frontend. For testing, a test account (universalcomputerwarehouse@gmail.com) was used to simulate all customer communications securely [39][41].  
  
**3. Successful response of request of /send\_order\_receipt API on POSTMAN**  
A screenshot of a computer

AI-generated content may be incorrect.

Screenshot of EMAIL triggered to the user after hitting the **API /send\_order\_receipt**

**A screenshot of a computer

AI-generated content may be incorrect.**

**4. Successful response of request of /send\_otp API on POSTMAN**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Screenshot of email sent to the user on email Valid for 5 mins  
  
A screenshot of a computer

AI-generated content may be incorrect.**

**Key Bugs & Fixes**

* **Bug**: Order status was not updating correctly in the track\_orders component.  
  **Solution:** Implemented dynamic status calculation based on the difference between the order date and the current date (Pending, Packaging, Delivered).
* **Bug:** Shelf capacity was being exceeded when placing supplier orders.  
  **Solution:** Added logic to prevent over-ordering by showing **toast warnings**, tracking cumulative quantity, and disabling the **‘Add’** button once the shelf reaches full capacity.
* **Bug:** Invalid or excessive product quantities were being added to the cart.  
  **Solution:** Input validation was implemented using Angular’s state tracking and custom error messages next to quantity fields.
* **Bug:** Session-based data like user email and order ID were lost across navigation or page reloads.  
  **Solution:** Used sessionStorage to persist data and validated its presence before order confirmation or email sending.
* **Bug:** Email receipt occasionally failed without clear feedback.  
  **Solution:** Added error handling and fallback alerts to inform users if email sending failed post-payment.

**Chapter 6: User Interface and Functionalities**

The Stock Control System was designed with a user-centric approach, prioritising clarity, responsiveness, and seamless functionality across roles. Built using Angular for the frontend and integrated with a Flask backend, the interface adapts dynamically based on user type Admin or Client. Each role is granted access to dedicated dashboards tailored to their responsibilities, whether it's order placement and tracking for clients or inventory oversight and analytics for admins. This chapter details the structure, navigation, and behaviour of these interfaces, explaining how user actions are connected to real-time backend operations via RESTful APIs.

**6.1 User Roles Overview**

The Stock Control System distinguishes between two primary user roles: **Clients (Customers)** and **Admins**, each with their own distinct interface and privileges. This separation not only enhances usability but also enforces security boundaries through role-based access control [44].

**Client Role**  
Clients access the system through a simplified interface focused on **product browsing, placing orders**, **tracking order status**, and **updating their profile**. Upon logging in, they are greeted with a dashboard displaying featured products, filters (e.g., low stock, best sellers), and a cart-based order mechanism refer to **Figure : Client Dashboard View** [45]. After selecting quantities, users proceed to a confirmation page and are sent a receipt via email post-payment.  
  
Fig Client Dashboard View  
  
A screenshot of a computer

AI-generated content may be incorrect.

Fig Thank You page  
  
A screenshot of a computer

AI-generated content may be incorrect.

**Admin Role**

Admins, on the other hand, gain access to a **multi-tabbed dashboard** which includes sections for **inventory control**, **warehouse zone management**, **supplier ordering**, **report generation**, and **sales forecasting**. This role is restricted to verified users and includes approval workflows during registration [46]. Admins can generate predictive insights using machine learning graphs, manage stock replenishments, monitor low-stock alerts, and apply discounts as needed.

Both roles are authenticated using JWT tokens, and permissions are enforced in both the frontend routes (Angular) and backend routes (Flask) [47]. The UI components across roles are built with **modular Angular components**, and page routing is dynamically rendered based on user type.

Fig Admin Dashboard Overview here

A screenshot of a computer

AI-generated content may be incorrect.

Fig Prediction graph of Products page

A graph with blue lines

AI-generated content may be incorrect.

**6.2 Client Dashboard Functionalities**

The **Client Dashboard** acts as the central interface for end-users to explore available products, filter based on stock levels or popularity, place orders, and track purchases. Built entirely with **Angular standalone components**, it integrates seamlessly with the Flask backend using RESTful API calls [45].

**Product Display and Filtering**

Clients are presented with a list of products fetched via GET /customer\_dashboard/products. Each product card displays the name, description, stock status, price, and a quantity input. The filtering system enables users to refine product listings based on **stock availability** (e.g., in stock, low stock), **price order**, and **bestseller rankings**, which is computed using the sold\_quantity attribute stored in the backend database [48].

Fig Insert Product Listing with Filters

A screenshot of a computer

AI-generated content may be incorrect.

**Cart Management and Order Placement**

Quantity input fields include validations to ensure users cannot select more than the available stock. A client can click "Add to Cart" and view selected items using the "View Basket" feature. Upon confirmation, items are passed to the Confirm Order page (/confirm-order) along with the user’s email stored in session storage.

This workflow involves a POST /customer\_dashboard/confirm\_order request which validates stock, calculates totals, and persists the order in the database. The order ID, total price, and item list are returned and displayed in the confirmation screen.

**Figure Insert Confirm Order Page  
  
A screenshot of a computer

AI-generated content may be incorrect.**

**Order Tracking**

After placing an order, clients can monitor its progress using the Track Orders section, which displays a timeline of order statuses including Pending, Packaging, Delivered status. This is powered by a dynamic calculation of date differences on the Flask backend, ensuring status transitions are time-based rather than manual [49].

A screenshot of a computer

AI-generated content may be incorrect.

**Profile Management**

Clients may update their **phone number and address** from the Update Profile section (/update-profile), which triggers a POST /customer\_dashboard/update\_account API. Email is used as a secure identifier, and updates are reflected immediately due to SQLAlchemy’s session commit logic.

A screenshot of a computer

AI-generated content may be incorrect.

The client interface is designed to be **mobile-responsive**, with **Bootstrap**-based layout, intuitive buttons, toast notifications for actions like adding to cart, and visually appealing product cards [35].

**6.3 Admin Dashboard Functionalities**

The **Admin Dashboard** is the control center of the Stock Control System, enabling administrators to manage products, monitor inventory, track profits, place supplier orders, and view detailed reports. It is structured using Angular child routes under /admin-dashboard, and integrates with the Flask backend using authenticated APIs [50].

**Inventory Overview and KPIs**The landing section of the admin dashboard which is DashboardHomeComponent presents high-level **Key Performance Indicators (KPIs)** such as total inventory, pending orders, low stock alerts, and top/least sold products. It also visualizes **daily profit trends over the last 10 days** using a GET /admin/overview API and Angular ApexCharts for dynamic charting [38].

**Warehouse Zone and Shelf Management**Each warehouse is divided into zones (A to D) with specific storage characteristics named like Zone A for accessories, Zone B for high-value items. Admins can navigate to each zone to view products, their shelf assignments, and current stock levels using the /admin/items\_by\_zone/<zone> endpoint.

A separate tab within this section displays shelf capacity usage, fetched from GET /admin/get\_shelves\_by\_zone/<zone>, visualized using bar charts. Shelf status is color-coded to indicate fullness, allowing proactive space management [32].

**Fig Zone & Shelf View**

A screenshot of a computer

AI-generated content may be incorrect.

**Supplier Orders and Validation**

Admins can place restock orders through the **Supplier Order** page. On selecting a supplier, available products are fetched and quantities can be specified. A real-time **validation logic** ensures that shelf capacity is not exceeded (POST /admin/validate\_shelf\_capacity), preventing warehouse overstocking.

Once validated, the admin submits all selected items to POST /admin/submit\_supplier\_orders, which stores the order in SupplierOrder and SupplierOrderDetails tables [51].

**Client and Supplier Order Reports**

Comprehensive reports are available under the **Reports** tab. Admins can filter **client orders** by date range, status, and search by Order ID using GET /admin/get\_client\_order\_reports. Similarly, **supplier orders** can be filtered by supplier name or date using GET /admin/get\_supplier\_order\_reports.

Results include details like number of items, total cost, order status, and date, aiding administrative decision-making.

**Fig Reports with export and search options:**

A screenshot of a computer

AI-generated content may be incorrect.

**6.4 Forecasting and Discount Management**

The forecasting and discount management features empower the admin to make data-driven restocking and promotional decisions based on predicted sales. These tools are tightly integrated with the backend machine learning model, allowing real-time insights into product performance over a future period.

**Sales Forecasting via Machine Learning**

Within the admin dashboard, administrators can access the **Sales Forecasting** module. By selecting a product from the dropdown, a request is triggered to the backend at the endpoint POST /admin/predict\_sales\_by\_product. This route uses the **Linear Regression model** trained on the past 15 days of historical sales data stored in the OrderDetails table. The trained model then predicts the next 30 days of daily sales volume for that specific product [1].

The predictions are returned in JSON format and rendered into an interactive **line chart using ApexCharts** within the Angular dashboard (dashboard-home.component.ts), enabling visual analysis of future demand [2].

**Applying Discounts to Least Sold Products**

Based on the sales insights, if a product is consistently underperforming (i.e., low predicted sales), the admin can mark it for a discount. The **discount management form** allows selection of a discount percentage (10%, 20%, or 30%) to be applied directly to a product via POST /admin/apply\_discount. The updated discount value is stored in the product model and automatically reflected in the client dashboard view [3].

To assist with decision-making, the **“Least Sold Products”** section on the dashboard displays bottom performers. Clicking on any of these products triggers the discount modal where the admin can confirm and apply the discount.