SMART PRICE OPTIMIZER USING MACHINE LEARNING

A PROJECT REPORT

Submitted by

NIRANJINI C	**811722104102**
NITHYA SREE D	**811722104105**
RAJESWARI M	**811722104118**

in partial fulfilment of the requirements for the award of the degree of Bachelor in Engineering

20CS7503 DESIGN PROJECT-3

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

K. RAMAKRISHNAN COLLEGE OF TECHNOLOGY

(AUTONOMOUS)

SAMAYAPURAM – 621 112

NOVEMBER 2025

K. RAMAKRISHNAN COLLEGE OF TECHNOLOGY

(AUTONOMOUS)

SAMAYAPURAM– 621112

BONAFIDE CERTIFICATE

The work embodied the project report titled **“SMART PRICE OPTIMIZER USING MACHINE LEARNING”** has been carried out by the students **NIRANJINI C, NITHYA SREE D, RAJESWARI M.** The work reported herein is original and we declare that the project is their own work, except where specifically acknowledged, and has not been copied from other sources or been previously submitted for assessment.

Date of Viva Voce: ……………………

Mrs. M. SURYA, M.E.,	**Mr. R. RAJAVARMAN, M.E.,(PH.D.,)**
SUPERVISOR	**HEAD OF THE DEPARTMENT**
Assistant Professor	Assistant Professor
Department of CSE	Department of CSE
K. Ramakrishnan College of Technology	K. Ramakrishnan College of Technology
(Autonomous)	(Autonomous)
Samayapuram – 621 112	Samayapuram – 621 112

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ABSTRACT

In today’s world, E-Commerce platforms have become an essential part of everyday shopping, offering convenience, competitive pricing, and wide product availability. However, most existing platforms follow static pricing strategies and provide limited post-purchase support information to customers. As a result, users often struggle to find the best real-time price and understand the long-term repairability or service options of the products they purchase. To address these limitations, this project presents SmartShop, an intelligent E-Commerce platform designed with dynamic pricing, personalized user benefits, automated negotiation, and innovative repairability insights that enhance decision-making for consumers.The system integrates Machine Learning-based dynamic pricing, which adjusts product prices automatically based on factors such as demand, competitor pricing, and stock availability. This ensures that customers receive fair, optimized prices while the platform maintains profitability.

Keywords: Dynamic Pricing, E-Commerce Platform, Service Center Recommendation, Pin-code Based Service Discovery, Google Maps API Integration, Machine Learning Integration.

ACKNOWLEDGEMENT

We thank our **Dr. N. Vasudevan**, Principal, for his valuable suggestions and support during the course of my research work.

We thank our **Mr. R. Rajavarman**, Head of the Department, Computer Science and Engineering for his valuable suggestions and support during the course of my research work.

We wish to record my deep sense of gratitude and profound thanks to my Guide **Mrs. M. Surya**, Department of CSE for her keen interest, inspiring guidance, constant encouragement with my work during all stages, to bring this thesis into fruition.

We are extremely indebted to our project coordinator **Mr. M. Saravanan**, Department of CSE, for his valuable suggestions and support during the course of my research work.

We also thank the faculty and non-teaching staff members of the Department Computer Science and Engineering, K Ramakrishnan College of Technology, Samayapuram, for their valuable support throughout the course of my research work.

Finally, we thank our parents, friends and our well wishes for their kind support.

SIGNATURE

____

____

____

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATION		FULL FORM
DRL	-	Deep Reinforcement Learning
ML	-	Machine Learning
UI	-	User Interface
ORM	-	Object Relational Mapping
MVC	-	Model-View Controller
RAM	-	Random Access Memory
UPI	-	Unified Payments Interface
IDE	-	Integrated Development Environment
REST API	-	Representational State Transfer Application Programming Interface

CHAPTER 1

INTRODUCTION

1.1 DESCRIPTION

In today’s digital world, E-Commerce has become an integral part of everyday life, transforming the way people purchase products and access services. With increasing internet penetration, smartphone usage, and digital payment adoption, online retail has expanded rapidly, offering convenience, speed, and a wider range of choices compared to traditional shopping. Modern consumers expect personalized recommendations, transparent pricing, fast delivery, and secure transactions. However, most existing platforms still rely on static pricing structures and generalized user experiences that do not fully adapt to changing market conditions or individual customer needs. As customer expectations continue to evolve, E-Commerce systems must adopt intelligent mechanisms to enhance affordability, trust, and personalization.

The concept of dynamic pricing has emerged as a powerful solution to address these challenges. Dynamic pricing allows online systems to adjust product prices in real time based on factors such as demand, competitor pricing, stock levels, and user behavior. Integrating machine learning algorithms further enhances decision-making by predicting optimal prices that balance customer satisfaction and business profitability. Despite advances made by leading platforms, many still lack transparent price optimization, user-specific discounts, and data-driven negotiation mechanisms. These gaps highlight the need for a smarter pricing approach that benefits both customers and sellers.

Additionally, modern shoppers increasingly value post-purchase support such as repairability information, service center accessibility, and fair maintenance costs-yet major E-Commerce websites do not provide such insights. Understanding repair difficulty, availability of spare parts, or nearest authorized service centers can influence a buyer’s confidence, especially when purchasing electronics. This absence of extended product information creates uncertainty and reduces user trust in long-term product value. Therefore, there is a strong need for an intelligent E-Commerce system that not only optimizes prices but also provides transparent, helpful, and user-centric information.

To address these issues, the SmartShop platform integrates dynamic pricing, loyalty-based offers, AI-driven auto-negotiation, and repairability scoring into a single unified system. The platform aims to provide a personalized shopping experience while ensuring fair pricing and greater transparency. By leveraging real-time data, machine learning, and user-specific insights, SmartShop enhances decision-making, improves customer satisfaction, and sets a foundation for the next generation of intelligent E-Commerce solutions.

1.2 Machine Learning for Dynamic Price Optimization

In today's world, where online shopping platforms are expanding rapidly, businesses face intense competition in attracting and retaining customers. Traditional E-Commerce systems rely on static or manually updated pricing strategies, which often fail to respond effectively to variations in customer behavior, market trends, competitor prices, and product availability. Due to these limitations, customers may either overpay for products or lose interest when prices are not competitive. To address this challenge, the domain of Machine Learning (ML) is adopted in this project, enabling a dynamic, data-driven pricing mechanism that optimizes product prices in real time. Machine Learning acts as the core engine of the SmartShop platform, predicting ideal pricing based on features such as base price, competitor price, stock availability, demand fluctuations, and seasonal purchase patterns. By training a RandomForestRegressor model, the system learns relationships between these attributes and computes an optimized price that offers maximum fairness to the customer and maximum profitability to the business.

The integration of ML allows the platform to adjust prices uniquely for each user based on loyalty level and purchase history, enabling hyper-personalized shopping experiences. This approach is significantly more intelligent than traditional pricing and is rarely implemented in typical online shopping applications. By integrating Machine Learning into the pricing mechanism, SmartShop establishes itself as an innovative, user-centric solution that mirrors real-world market dynamics and adapts instantly to ongoing changes.

1.3 Web Development & E-Commerce System Architecture

E-Commerce platforms today require sophisticated system architecture to handle multiple user interactions such as product browsing, cart management, secure checkout, and personalized dashboards. This project belongs strongly to the domain of Full-Stack Web Development, integrating frontend, backend, and database layers into a cohesive platform. Smartshop is implemented using Flask, a lightweight yet powerful Python web framework, which provides flexibility in routing, authentication, templating, and server-side processing. The backend implements user registration, login authentication, dynamic pricing logic, cart operations, order management, and admin functionalities. SQLAlchemy ORM manages communication with an SQLite database, enabling structured storage and retrieval of products, users, orders, and cart items. The frontend is developed using HTML, CSS, Bootstrap, and Jinja2 templates, ensuring a responsive, modern interface where users can seamlessly interact with the platform. The system architecture follows an MVC-inspired structure, where the backend handles logic, models manage data, and templates handle presentation. Advanced features like loyalty-based discounts, cashback systems, auto-negotiation pricing, and product-specific pages make Smartshop significantly more interactive than traditional static E-Commerce sites. Additionally, the admin panel enables product management, stock updates, and competitive pricing entry, offering a complete business dashboard experience. Through effective combination of UI design and backend engineering, Smartshop provides a secure, user-friendly marketplace that elevates customer experience while ensuring high performance and maintainability of the system. This domain emphasizes how modern web technologies, when integrated effectively, can power a scalable and interactive E-Commerce application suitable for real-world deployment.

CHAPTER 2

LITERATURE SURVEY

2.1 Transfer Learning Based Adaptive Automated Negotiating Agent in E-Commerce (Sengupta et al., 2022)- Ayan Sengupta, Shinji Nakadai, and Yasser Mohammad

A transfer-learning based automated negotiating agent for E-Commerce scenarios where buyers and sellers bargain over price and other terms. Instead of fixed rules, the agent learns negotiation strategies from previous interactions and then adapts them when moving to new markets or products. The paper shows that using transfer learning significantly reduces training time while still achieving good negotiated outcomes, such as mutually acceptable prices and higher overall utility for both sides. This work directly supports your idea of an "auto negotiation" feature where the system proposes a better price if the user enters a counter-offer. Traditional negotiation bots follow fixed rules ("offer 10% discount, then 5%, etc."), but this AI agent learns from previous negotiation outcomes and adapts. When transferred to a new product category, the model doesn't need to relearn from scratch-transfer learning allows it to reuse prior knowledge. The study proves that negotiations led to better customer engagement and satisfaction, as users felt more control over pricing. This directly relates to Smartshop's auto-negotiate feature: your system currently uses stock, demand, and loyalty to produce an automated discount. In future, a negotiation history table can track user counter-offers and gradually train a model to offer smarter discounts. This paper gives strong theoretical backing to show in your report.

The study also emphasises transparency and fairness. A negotiation engine must not always push the lowest price possible, as this could harm business margins. Instead, it should maintain a balance between competitiveness and profitability. The adaptive model used in the paper ensures controlled discounting while maintaining customer satisfaction. This makes the negotiation process context-aware and enhances the overall decision-making capability of the platform. For Smartshop, this integration can mean combining negotiation outcomes with pricing predictions.

2.2 Deep Reinforcement Learning for Personalized Dynamic Pricing in Retail (2024)- David Brown and Mike Olumide.

The concept of dynamic pricing has evolved significantly with the advancement of machine learning, and recent research highlights the impact of Deep Reinforcement Learning (DRL) in this domain. This study focuses on how DRL can be used to set personalized, context-aware prices in modern retail environments. Traditional pricing strategies rely heavily on fixed markups, rule-based adjustments, or periodic manual revisions. While these methods are simple to implement, they fail to capture demand uncertainty, changing customer behavior, and real-time market influences. The 2024 study addresses these gaps by proposing a DRL agent capable of making pricing decisions through continuous interactions with the environment. The DRL model treats the pricing environment as a sequential decision-making problem. States provided to the agent include historical sales records, current inventory level, time of day, seasonal factors, and customer segmentation. These rich contextual inputs allow the agent to understand demand patterns, identify trends, and anticipate customer reactions.

Based on the observed state, the agent selects an action typically a price adjustment-that is expected to maximize long-term revenue. Unlike rule-based strategies that focus on short-term gains, the DRL framework priorities cumulative rewards over multiple time steps. The authors use simulated and real-world retail datasets to train the agent. Over thousands of episodes of interaction, the DRL algorithm learns price adjustment patterns that lead to higher overall revenue. The study demonstrates that DRL-based strategies consistently outperform static or heuristic-based methods, especially in environments where demand fluctuates unpredictably. This includes scenarios like festive seasons, flash sales, and sudden competitor price changes. The study also highlights the need for explainability in dynamic pricing systems. Transparent communication of why a price was changed helps reduce user distrust and increases the acceptance of AI-generated pricing.

2.3 Artificial Intelligence for Pricing in E-Commerce (2024)-Gavade

The 2024 article “Artificial Intelligence for Pricing in E-Commerce” by Gavade and co-authors presents a comprehensive framework for how AI can be used to manage pricing in modern online marketplaces. The authors describe pricing as a multi-layered decision-making process that cannot rely only on prediction models. Instead, it requires an integrated system that connects forecasting, segmentation, optimization, promotion rules, and even inventory and recommendation engines. This makes pricing a central part of the entire E-Commerce ecosystem rather than an isolated function. The study introduces a modular AI-based architecture with three interconnected layers: prediction, segmentation, and optimization. The prediction layer focuses on estimating key variables such as demand, competitor price trends, customer interest, and inventory turnover. These predictions are typically produced using machine learning models trained on sales history, market fluctuations, and seasonal behavior. The segmentation layer classifies customers based on demographics, purchase patterns, loyalty level, and browsing history. This helps businesses personalise prices and promotional messages for different user segments. Finally, the optimization layer uses the outputs from the first two layers to compute the optimal price for every product. It considers both internal factors (cost, stock level, profitability goals) and external factors (competitor actions, seasonal campaigns, and customer demand signals).

A key insight from the paper is that pricing is not limited to numerical calculations-it must work in harmony with other operational modules of an E-Commerce platform. For example, recommendation systems can influence pricing by directing users towards high-margin or overstocked products. Similarly, inventory management systems depend on accurate pricing signals to decide when to restock or clear items. Marketing campaigns-including flash sales, seasonal promotions, or loyalty rewards-also rely on pricing decisions generated by AI engines. This interconnected nature means that the pricing engine must be flexible, dynamic, and capable of working with real-time data. The authors provide detailed examples of how demand forecasting is used to refine pricing decisions.

2.4 Personalization and Customer Loyalty in E - Commerce Pricing Gotmare, 2021

Gotmare (2021) analyses how personalized pricing and offers influence customer loyalty and purchase intent in E-Commerce. The study surveys online consumers and finds that personalized deals-such as special discounts for repeat customers, targeted coupons, and tailored price reductions-can significantly improve satisfaction and loyalty if they are perceived as fair and transparent. However, when personalization is too opaque or seems discriminatory, it can hurt trust and lead to negative word-of-mouth. This is a direct implementation of what the paper suggests: segment users based on their relationship with the platform and apply different benefits to encourage repeat purchases. The literature also suggests clearly communicating why a user is getting a particular offer (e.g., “loyalty discount”), which you can highlight in the UI using badges and tooltips. This research highlights how tailored offers create emotional connection and loyalty between customers and platforms. Personalized pricing includes exclusive discounts, recommendations, early access sales, and cashback, leading to higher lifetime value per user.

The study also raises a caution: customers may view aggressive personalization as unfair if not explained clearly. It emphasizes the importance of “explainable personalization”, where platforms tell the user why they received a certain offer. Smartshop does this by labeling discounts as new user discount, loyalty discount, cashback, etc. The paper also suggests tracking a user’s total spends and purchase frequency, which matches your fields total spent, orders count, and cashback balance. This literature validates these design decisions strongly. For Smartshop, this integration can mean combining negotiation outcomes with pricing predictions (you optimize price function) to produce more realistic discounts. For example, if demand for a product is rising, the negotiation model could limit discounts even if users negotiate aggressively. Similarly, if the product is overstocked, the model could allow deeper discounts. Finally, the study highlights that negotiation agents can significantly increase user interaction time on the platform.

2.5 Deep Q-Learning Framework for Dynamic Pricing in E-Commerce (2025) - Jiancai Li and Biaoxin Chen.

The 2025 study on Deep Q-Learning (DQN) for dynamic E-Commerce pricing introduces a highly advanced reinforcement learning framework designed to autonomously adjust product prices in response to continuously changing market conditions. Unlike traditional statistical or machine learning models that merely predict optimal prices using fixed datasets, DQN treats pricing as a sequential decision-making problem, where each pricing action influences future sales outcomes, customer behavior, and inventory levels. In this framework, every decision is modeled as an action taken by an intelligent pricing agent, while the state representation includes multiple real-time inputs such as inventory stock level, time of day, seasonality, competitor price changes, click-through rate (CTR), user engagement signals, ongoing offers, and recent sales momentum. The reward function is defined not only by immediate profit but also by longer-term utility, such as stock utilization efficiency, conversion rate stability, and customer retention measures. The authors demonstrate that Deep Q-Learning captures complex price-demand relationships that rule-based systems cannot learn, especially under conditions of uncertainty, shifting competition, and sudden demand shocks. Over multiple simulated episodes involving thousands of pricing interactions, the DQN agent gradually learns pricing strategies that outperform static price tags, linear regression models, and even traditional markdown policies.

The study reveals that Deep Q-Learning autonomously discovers intelligent behaviors such as increasing prices when competitors are stock-out, offering limited-time discounts during low-demand periods, lowering prices strategically when inventory is high, and adjusting pricing patterns based on different customer segments. This research has high relevance for your Smartshop project. Currently, Smartshop implements a hybrid pricing system combining a Random Forest Regressor for base price predictions and rule-based business logic layers for discounts, new-user offers, loyalty benefits, and negotiation adjustments.

2.6 Product Recommendation with Price Personalization in E-Commerce (Mahdavian et al., 2025)

Mahdavian et al. (2025) present a product recommendation system that also personalizes the displayed price for each user. Instead of treating recommendations and pricing as separate problems, they jointly optimize both: the model suggests products that match user preferences and simultaneously adjusts the offered price within allowed business constraints. The system considers user history, browsing behaviour, and price sensitivity to decide how much discount or markup to show, leading to higher click-through and conversion rates compared to non-personalized setups.

This approach is very close to what you are doing conceptually: your get_user_type() + apply_offers() logic already personalizes final price based on user behaviour (orders_count, total_spent) and loyalty category. You could later extend your app by adding a “You may also like” section on the product page and in the cart, where recommended products already show the dynamic final price for that user. This paper supports the idea that combining recommendations with pricing can give better user experience and higher revenue than keeping them separate, and it justifies your design choice of linking loyalty, cashback and pricing all together in Smartshop. This paper proposes a joint optimization system that makes product recommendations while also personalizing the displayed price.

This results in a dynamic and highly personalized shopping experience. The authors show that conversion rates increased drastically when price personalization was integrated into recommendation lists. Smartshop already adjusts prices based on user type and loyalty; adding recommendation integration with personalized prices would align your app with this advanced model.

2.7 Dynamic Pricing Strategies Implementing Machine Learning Algorithms in E-Commerce (2024)- Sheed Iseal and Michael Halli

Dynamic pricing has become an essential component of modern E-Commerce platforms, and this 2024 study provides a detailed investigation of how machine learning algorithms can be used to continuously update product prices. Unlike traditional pricing methods that rely on fixed formulas or human intuition, this paper demonstrates that data-driven models can understand complex market patterns and make more accurate price recommendations. The authors analyse different machine learning techniques and evaluate how effectively they predict optimal selling prices based on real-world data. The study begins by identifying key factors that influence price decisions in E-Commerce. These include historical sales trends, competitor prices, product categories, seasonal effects, promotional periods, and inventory status. The research emphasises that these variables often interact in non-linear ways, making simple linear models insufficient. To address this, the authors explore a wide range of algorithms, such as Linear Regression, Random Forest, Gradient Boosting, XGBoost, Support Vector Regression, and Decision Trees.

One of the main contributions of the paper is a thorough comparison of these algorithms. Using large datasets collected from online retail platforms, the authors train each model to predict the price that maximises revenue while staying within business constraints. Their experiments show that tree-based models and ensemble methods outperform traditional statistical techniques. For example, Random Forest and XGBoost provide more stable results because they can capture non-linear relationships, interactions between variables, and the noisy nature of competitor data.The research also highlights the importance of regression-based approaches for dynamic pricing. Linear Regression is simple to implement and provides interpretability, but it often struggles when the data contains outliers, sudden spikes in demand, or irregular competitor price movements. On the other hand, Random Forest models can handle missing data, noisy features, and variable importance ranking, making them more suitable for real-world pricing tasks.

2.8 AI and Dynamic Pricing in E-Commerce: Strategies for Optimizing Revenue (2023)- S. Vimaladevi, V. Gopi, M. Suresh, M. Rajalakshmi

The 2023 paper titled "AI and Dynamic Pricing in E-Commerce: Strategies for Optimizing Revenue" offers a holistic, managerial, and strategic understanding of how artificial intelligence transforms pricing decisions in online retail. Unlike technical studies that focus primarily on algorithms, this research provides a broad framework that combines business strategy, customer psychology, fairness considerations, and real-world implementation challenges. It explains how AI-driven systems can integrate cost-based, competitor-based, and demand-based pricing strategies into one unified approach that responds to market shifts in real time. The authors begin by categorizing pricing methods into three core groups. The cost-based strategy involves setting prices by adding a margin to the product's cost. This approach provides stability but does not react to competitors or demand changes. The competitor-based strategy adjusts prices based on market competition, ensuring that products remain attractive relative to alternatives. The demand-based strategy, on the other hand, uses indicators such as customer interest, traffic level, conversion rate, and stock movement to determine how much customers are willing to pay. The paper argues that AI can combine these strategies effectively by evaluating multiple signals simultaneously.

To achieve this integration, the authors propose a three-layer AI system consisting of forecasting, segmentation, and optimization. The forecasting layer predicts demand, competitor behavior, and inventory turnover using machine learning models. The segmentation layer classifies customers into behavioral groups such as loyal customers, impulse buyers, bargain hunters, and occasional visitors. Each group responds differently to price changes, making segmentation a key step in personalizing the pricing experience. The optimization layer determines the best price, taking into account predicted demand, business rules, customer segment, and fairness constraints.

The study places strong emphasis on real-time data streams, explaining how prices can be updated dynamically based on live signals such as page views, cart abandonment rates, stock availability, and competitor price crawlers. The authors highlight real-world examples from Amazon, Walmart, and airline booking platforms, where prices may update every 5 minutes-or even every 30 seconds-leading to significant revenue improvements.

2.9 Repairability Index and Users' Intention to Repair Smartphones - Torca-Adell et al., 2025

Torca-Adell et al. (2025) investigate how the introduction of a clear and standardized repairability index influences consumers' decisions to repair or replace their smartphones. Their work is inspired by global trends such as the French Repairability Index law, the EU's Right-to-Repair policies, and rising consumer awareness about sustainability. The authors examine several psychological and practical factors that determine whether a user chooses to repair: perceived usefulness of the repair score, simplicity of understanding the index, trust in the organization providing the information, availability of spare parts, clarity of repair cost estimates, and environmental attitudes of the buyer. Their findings reveal that repairability information has a strong behavioural impact, especially when it is presented in a very simple form-typically a 0–10 score, colour-coded, accompanied by icons or quick messages. Participants in their experiment were far more likely to consider repairing when repairability scores were shown directly on product pages, compared to when no such information was available. The clarity of presentation also mattered significantly; users preferred minimal but meaningful indicators such as "Repair Score: 7/10 (Easy to repair), Battery replaceable, Screen cost low)" rather than long, technical descriptions.

The study highlights that trust is a crucial mediating factor. If the repairability score is assumed to come from a credible and neutral source-such as an independent agency or the manufacturer-users are more confident about making repair decisions. When trust is low, even a high repairability score fails to influence behaviour. The research also explores how repair costs, when shown transparently, strongly affect behavioural intention. Users are much more likely to repair their phones when the cost of repair is visibly lower than buying a new model, and when spare parts are available at reasonable prices. Conversely, if parts are expensive or unavailable, or if repair difficulty (measured in time or skill level) is high, users tend to replace rather than repair. Torca-Adell et al. also studied the influence of environmental consciousness. Users with strong environmental attitudes were highly responsive to repairability scores, suggesting that eco-friendly shopping portals can leverage this information to attract sustainability-focused buyers.

2.10 Location-Aware Point-of-Interest Recommendation-Zeng et al., 2020

Zeng et al. (2020) present a comprehensive study on location-aware point-of-interest (POI) recommendation systems that help users discover relevant places based on geographical proximity, personal preferences, mobility patterns, and contextual information. Their model integrates multiple layers of intelligence rather than relying only on the nearest-distance assumption. The authors show that simple distance-based methods often fail to capture the true relevance of a POI because users' choices depend on behavioural patterns-such as which kinds of places they visited before, how far they are willing to travel, what category of POI they prefer at a given time, and the user's daily activity context. The study proposes a hybrid recommendation framework that incorporates geographic influence (using distance decay functions), temporal patterns (time-of-day and weekday effects), user check-in history, categorical preferences, and latent factor models. Their experiments demonstrate that incorporating context significantly improves POI prediction accuracy and enhances user satisfaction because the system does not merely suggest the closest place, but the most relevant one. Zeng et al. also emphasize the value of continuously learning from user interactions: when users frequently visit or skip certain POIs, the model gradually adapts its ranking. This makes the system smarter over time and more aligned with real-world user behaviour.

Although Zeng et al.'s work is primarily applied to restaurants, tourist spots, shopping centers, and entertainment venues, the underlying concepts transfer extremely well to an E-Commerce environmen2.1t where the goal is to recommend nearby repair or service centers. In your SmartShop system, the same logic that recommends nearby POIs can be adapted to create a "Nearby Service Center Finder" for electronic devices. Instead of recommending food or entertainment locations, the system would recommend authorized service centers, brand-specific repair shops, and certified third-party repair providers. Zeng's emphasis on combining distance with relevance directly supports your idea that recommending service centers should not be based only on proximity; relevance must include the product's brand, the category of repair needed, user urgency, and ratings or reliability of the center. For example, if a user purchases a Samsung smartphone on SmartShop, the recommendation engine should prioritize Samsung-authorized service centers, even if a generic repair shop is slightly closer.

CHAPTER 3
EXISTING SYSTEM

In today’s world, most E-Commerce platforms such as Amazon, Flipkart, Myntra, and Ajio follow a well-structured multi-tier system architecture that helps them manage millions of products, users, and orders efficiently. These systems typically rely on a combination of a user interface layer, business logic layer, and database layer to provide a smooth shopping experience. The user interface layer is what customers directly interact with, which includes web applications built using HTML, CSS, JavaScript, or frameworks like React and Angular, as well as native Android and iOS apps. Through this interface, customers can browse products, view details, add items to the cart, check their orders, and make payments. All these interactions are processed by the application layer, which contains the business logic that controls essential operations like user authentication, product management, search filtering, inventory updates, cart handling, order placement, returns, refunds, and personalized recommendations. This layer also integrates advanced search technologies such as ElasticSearch to help users find products quickly and applies fixed pricing strategies that are usually manually managed or seasonally updated.

The business logic communicates with the database layer, which stores massive volumes of information such as user profiles, product details, inventory status, order history, reviews, delivery addresses, payment records, and transaction logs. These databases are built using technologies like MySQL, PostgreSQL, MongoDB, or DynamoDB and are optimized for fast access and secure storage. Modern E-Commerce systems also use a microservices architecture, where individual features such as cart service, payment service, search service, user service, and order service run independently and communicate through APIs. This allows companies to scale each component separately and push updates without affecting the entire system. To deliver product images and static files quickly across different regions, they also rely heavily on Content Delivery Networks such as Cloudflare or AWS CloudFront. Security is another crucial part of the existing architecture; platforms use SSL encryption, secure

OTP validation, multi-factor authentication, fraud detection algorithms, and compliance with global payment security standards to protect user data and online transactions.

The logistics and delivery management system is responsible for order shipping, warehouse management, delivery partner assignment, real-time order tracking, automated notifications, and route optimization. This part interacts with courier partners such as Ekart, Delhivery, BlueDart, and Ecom Express. However, despite being advanced, current E-Commerce platforms come with certain limitations. Most systems follow fixed or seasonal pricing strategies and do not implement real-time dynamic pricing based on demand or stock. They do not offer negotiation features that allow users to bargain for a better price, nor do they provide repairability scores, spare-part availability details, or service center information. Customers rarely get transparency about how repairable a product is or the cost involved in future maintenance. These limitations highlight the gap between user needs and the existing market solutions, creating opportunities for next-generation features such as AI-based dynamic pricing, auto-negotiation, repair prediction systems, and real-time service center locators.

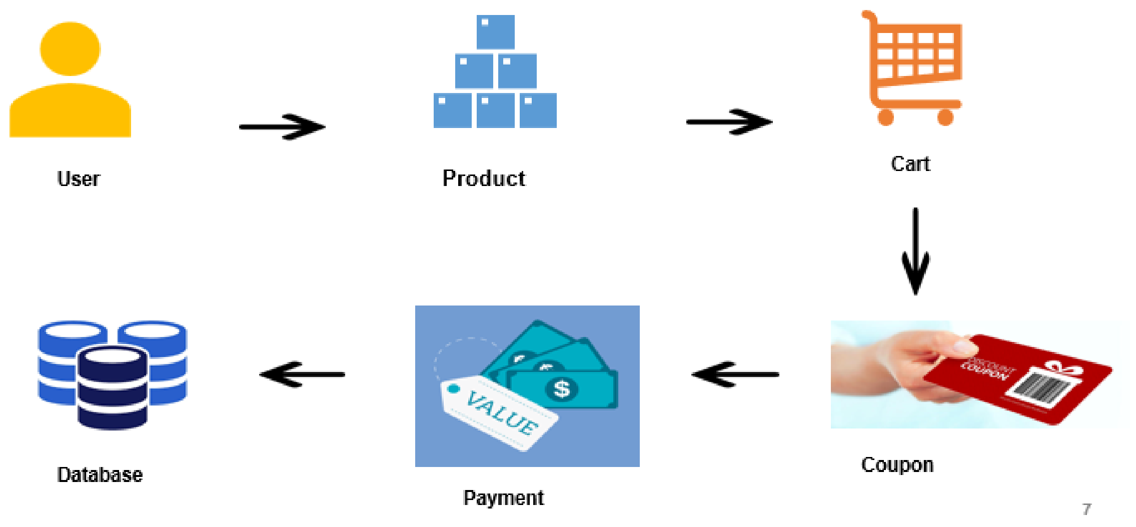


FIG.3.1 Existing System Diagram

CHAPTER 4

PROBLEMS IDENTIFIED

Most existing E-Commerce platforms offer the same discounts to all customers, without evaluating loyalty, purchase frequency, or historical spending. This leads to low customer retention, as loyal users do not receive personalized benefits that reward their consistent engagement. Businesses often provide blanket discounts to all users during sales seasons. Since these discounts are not optimized based on customer behavior, demand, or price sensitivity, it results in excessive revenue leakage and lowers overall profit margins. Traditional pricing strategies rely heavily on human decision-making. For large inventories, this becomes inefficient because manual adjustments are slow, inconsistent, and can lead to pricing mismatches, human errors, or outdated prices during peaks.

Existing systems do not update pricing instantly based on real-time fluctuations such as demand spikes, stock variations, user preferences, or competitor changes. The inability to react dynamically leads to missed sales opportunities and poor competitiveness. Most online marketplaces follow a fixed-price model with no scope for negotiation. This eliminates human-like bargaining flexibility and fails to satisfy price-conscious customers who expect personalized deals or negotiated offers. Current platforms provide minimal data about repairability, maintenance cost, or long-term sustainability of products. Users cannot estimate the total lifetime cost or locate nearby authorized service centers, resulting in uninformed purchase decisions. Existing systems do not provide ML-driven personalized pricing. Without analyzing user behavior, browsing patterns, or purchasing power, they fail to offer context-aware price recommendations, reducing conversion rates.

Customers usually only see the final price, without understanding demand effects, stock levels, or available discount opportunities. This lack of transparency causes distrust and results in abandoned carts or reduced user satisfaction. Product pages in most platforms contain only a short, generic description. They do not show technical specifications, usage instructions, repair cost, or condition-based dynamic insights. This leaves customers with incomplete knowledge about the product's actual value. Existing E-Commerce platforms usually do not integrate user address or pincode to suggest nearby service centers, stock availability, delivery predictions, or repair locations. This limits post-purchase support, reducing overall customer experience and trust.

In today’s rapidly evolving digital marketplace, traditional E-Commerce platforms such as Amazon and Flipkart primarily focus on fixed pricing strategies that do not adapt to real-time market factors or individual user behavior. As a result, customers often end up paying more than necessary, and sellers struggle to optimize profits while maintaining customer satisfaction. Additionally, current platforms lack transparency in product repairability, long-term maintenance costs, and access to nearby service centers-factors that greatly influence a customer’s purchase decision but are often ignored. Users also face limited personalization, as offers and pricing are not tailored to their loyalty or purchase history. The absence of automatic price negotiation further reduces customer engagement and prevents buyers from obtaining fair deals based on supply, demand, and product stock levels. Furthermore, most platforms provide basic cart and checkout systems but do not assist users with warranty predictions, product lifespan insights, or post-purchase support information. These gaps highlight the need for a smarter, more adaptive E-Commerce solution. Therefore, the problem lies in the lack of a dynamic, intelligent, and transparent shopping platform that uses machine learning to adjust prices, enhance user experience, and provide decision-support features that simplify purchasing, maintenance, and long-term product management.

CHAPTER 5

PROPOSED SYSTEM

The proposed system introduces an intelligent E-Commerce architecture that goes beyond traditional online shopping platforms by integrating AI-driven pricing, automated negotiation, real-time repairability scoring, and nearby service-center discovery into a single unified system. The architecture begins with an enhanced user interface layer built using Flask, Bootstrap, and responsive design principles, allowing customers to seamlessly browse products, view dynamic prices, negotiate with the system, and access repair-related insights on each product. The application communicates with the backend through REST APIs that manage authentication, pricing logic, negotiation algorithms, and service center lookup. Unlike existing platforms that use fixed or seasonally updated prices, this system integrates a machine learning model that analyzes multiple factors-such as competitor pricing, stock availability, product demand, user loyalty score, and historical purchase patterns-to compute an optimized product price in real time. The negotiation module further enhances the experience by allowing users to request a lower price, which is calculated dynamically based on inventory conditions, user history, and predefined business rules. This creates a personalized and adaptive pricing mechanism that does not exist in current E-Commerce architectures.

The proposed architecture also incorporates a repairability and service-support layer, which assigns each product a repairability score based on factors like component availability, estimated repair cost, and historical failure rates. The system connects to external APIs such as Google Maps or OpenStreetMap to fetch real-time nearby service centers using the user's pin code or geolocation. This allows users to not only view the technical details of a product but also understand its long-term maintenance feasibility-something modern E-Commerce platforms currently lack. On the backend, the system uses an extended database schema to store user profiles, addresses, loyalty points, cashback balance, and negotiation history. The product table includes additional attributes such as repairability score, spare part availability, average repair cost, and associated service centers.

A dedicated microservice-like structure is simulated within the Flask backend: separate modules handle pricing, negotiation logic, repairability analytics, user management, and payment processing. The payment system supports multiple modes such as credit card, debit card, UPI, and wallet-based transactions. After successful payment, the order management module updates inventory levels, assigns cashback rewards, logs purchase behavior, and updates the user's loyalty tier. All static resources such as product images are served from the server's static directory, while dynamic product data flows through the optimized API pipeline. The architecture enforces strict security through hashed passwords, role-based admin access, and secured communication for critical functions like password resets via email. Altogether, the proposed system architecture creates an intelligent, context-aware, and highly adaptive shopping platform that solves the limitations of existing E-Commerce systems by blending AI-powered pricing, interactive negotiation, and repair transparency into a modern, scalable, and user-centered solution.

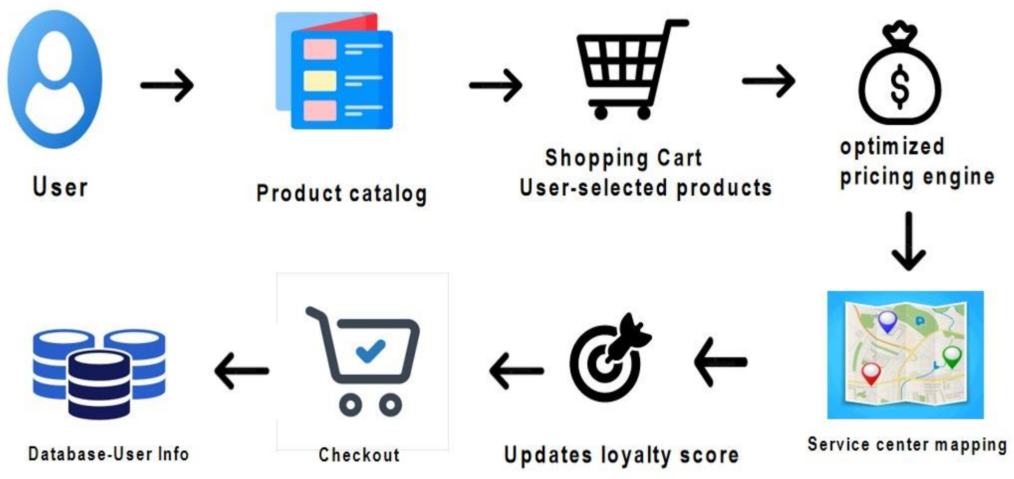


FIG. 5.1 Proposed System Diagram

CHAPTER 6

SYSTEM REQUIREMENTS

6.1 HARDWARE REQUIREMENTS

Component	**Specification**
Processor	Intel i5 / RMD Ryzen 5 or above
RAM	Minimum 8 GB
Storage	At least 200 MB free space
Mode	Smartphone / Laptop / Desktop that supports modern browsers
Operating System	Windows / Linux / macOS

6.2 SOFTWARE REQUIREMENTS

Component	**Specification**
Operating System	Windows 10 or higher (for COM automation)
Programming languages	Python 3.9+,HTML5,CSS
Python Libraries	Opencv-Python, Mediapipe, Numpy, Werkzeug, Flask
IDE	Visual Studio Code / Python Extension
	SQLite Viewer extension
Database	SQLite (Local development)SQLAlchemy ORM

CHAPTER 7

SYSTEM IMPLEMENTATIONS

7.1 LIST OF MODULES

- User Management Module
- Product Management Module
- Auto Negotiation Module
- Repairability and service center mapping Module
- Cart and Checkout Module

7.2 MODULE DESCRIPTION

7.2.1 User Management Module

The User Management Module is responsible for handling all user-related operations within the system. It begins with user registration, where users enter their personal details such as name, email, password, phone number, address, and pincode. Validation checks ensure that names contain only alphabets, passwords meet complexity rules, and emails are unique. All passwords are hashed using a secure hashing algorithm before being stored in the database, ensuring user security and protection against unauthorized access. Once registered, users can log in using their credentials, and the system creates a session to maintain user state across pages. Flask-Login manages user sessions securely, preventing unauthorized users from accessing exclusive features like cart, negotiation, and checkout.The module differentiates between normal users and admin accounts, with admin users having access to product management functionalities. Forgot Password functionality is also part of this module. When a user forgets their password, they can request a reset link by entering their registered email. The system generates a time-sensitive token and sends a password reset link to their email using Flask-Mail. When the user clicks the link, they are redirected to a password reset page where they can set a new password.

7.2.2 Product Management Module

The Product Management Module handles all operations related to adding, updating, displaying, and managing products in the system. This module is primarily accessible to admin users, ensuring that only authorized personnel can modify product details. Products contain several important attributes such as name, description, base price, stock quantity, category, image, demand level, competitor price, repairability score, and estimated repair cost. These attributes are stored in the database and used by various other modules such as dynamic pricing, negotiation, and repairability prediction. Admins can upload product images, update stock levels, and modify prices based on market trends.The frontend product listing page retrieves product information from the database and displays it to the user with optimized UI styling. Each product has a dedicated details page where more information is shown, including dynamic pricing results, detailed description, repairability score, difficulty level, and service availability. Product descriptions, previously shown in a single line, have now been expanded into multiple lines to provide more clarity about features, specifications, and technical details. This helps customers make better purchase decisions and reduces confusion.

7.2.3 Auto Negotiation System Module

The Auto Negotiation Module is a unique feature that simulates human-like bargaining within the E-Commerce system. When users click the “Auto Negotiate” button on a product page, the system calculates an optional additional discount based on multiple criteria. Inputs such as optimized price, user loyalty status, stock quantity, and product demand are considered. If demand is low or stock is high, the system offers a higher negotiation discount. Loyal users are given even better negotiation outcomes, simulating real-world bargaining where frequent buyers get better deals.The system ensures that negotiation discounts remain within business limits, preventing excessive price drops. The negotiation result is returned as JSON and dynamically displayed to the user with the final negotiated price, total discount, and extra deduction percentage. This provides transparency and enhances user experience by showing exactly how negotiation affects price.Auto negotiation creates a feeling of interaction and engagement, making the shopping process more enjoyable. It gives users a sense of control and satisfaction when they successfully reduce prices. Businesses also benefit because negotiation is calculated intelligently rather than randomly. It prevents giving the same negotiation discount to every user, protecting profit margins.This module also increases conversion rates by persuading hesitant users to complete a purchase after receiving a better deal. It works seamlessly with loyalty and dynamic pricing modules to generate fair, balanced, and user-specific negotiated prices.

7.2.4 Service Center Mapping Module

The Repairability Module provides users with essential long-term product maintenance information before purchasing. Each product is assigned a repairability score, difficulty level, and approximate repair cost. These values help users understand how easy or difficult it is to repair the product in the future. This is important because users often overlook repair costs and regret purchases later. The module helps them make informed decisions by being transparent about repair expectations.The system also integrates Google Maps to fetch nearby service centers based on the user’s pincode stored during registration. When viewing a product, the backend retrieves the user’s location and displays authorized or third-party repair centers on an embedded Live Google Map. This makes the system more interactive and useful, especially for electronic products where service locality matters.Repairability scores are displayed clearly on the product page, along with explanations and estimated repair costs. Users can also get directions to the nearest service center directly through Google Maps. This improves convenience and reduces the time users spend searching for repair options manually.This module improves the overall reliability of the system and increases user trust because it shows long-term care details instead of just selling products. It differentiates your platform from typical E-Commerce sites by considering the post-purchase lifecycle of products.

7.2.5 Cart and Checkout Module

The Cart Module allows users to add products they want to purchase and calculate total costs in real time. Each item is displayed with its name, optimized price, loyalty discount, and final payable price. Users can add, remove, or update quantities inside the cart. The backend refreshes totals automatically after every update. This ensures accurate cost calculation based on real-time dynamic pricing.During checkout, users choose a payment method such as UPI, Google Pay, debit card, or net banking. The system calculates total payable amount including all applied discounts, cashback, and negotiated amounts. After the user confirms their payment, an Order is created in the database. OrderItem entries are also stored to maintain what items were included in that purchase. Stock is reduced according to purchase quantity, and the user's loyalty information gets updated instantly.

CHAPTER 8

SYSTEM TESTING

8.1 Unit Testing

Unit testing is the foundational level of software testing and is essential for ensuring that the smallest components of the application function correctly in isolation. In this project, which involves dynamic pricing, auto-negotiation, user authentication, repairability score, and database interaction, unit testing plays a significant role in validating each function and method independently. For example, the pricing engine contains functions such as optimize_price(), apply_offers(), and auto_negotiate(), each of which must independently process inputs and return accurate outputs. Any errors in these functions would propagate throughout the system, affecting product pricing, final billing, and user satisfaction.Unit testing ensures that every computation-such as discount calculation, cashback estimation, loyalty scoring, and the application of negotiation logical performs correctly. Additionally, unit testing is used to validate edge cases, such as extremely high demand values, low stock quantities, or incorrect user types. Testing such conditions guarantees that the algorithm functions correctly in real-world scenarios.Unit tests also validate user authentication functions including registration, password hashing, login verification, and token generation for the "Forgot Password" functionality. Similarly, repair score retrieval and Google Maps API handlers must correctly return structured data.

8.2 Integration Testing

Integration testing focuses on verifying whether different modules of the application work together seamlessly. In this project, several important modules interact: the dynamic pricing model interacts with the product display system, cart system, negotiation system, checkout module, and order management. Each module functions independently, but they must operate cohesively when combined.For instance, when a user views a product, the optimized price must be calculated using machine learning logic. When the product is added to the cart, the same price must be updated consistently. During checkout, discounts, cashback, loyalty updates, and stock reduction must all operate jointly. Integration testing ensures these interconnected workflows behave correctly under various conditions.Another important integration scenario is repairability score fetching and service-center verification based on user pincode. The user registration module stores the pincode, the product module stores repair score details, and Google Maps API retrieves nearby service centers. Integration testing ensures all these modules exchange data properly.Payment processing is another critical integration area. The system should correctly handle payment method selection (UPI, debit card, credit card), update order records, clear the user's cart, and generate a success message.Integration testing exposes issues such as API failures, inconsistent data flow, database synchronization problems, missing foreign keys, or incorrect UI updates.

8.3 Functional Testing

Functional testing verifies whether the system meets all functional requirements and performs tasks exactly as specified. In this project, where many user-facing features are involved, functional testing is crucial to ensure the application behaves normally under different user actions.Key functionalities include user registration, login, logout, viewing products, browsing categories, checking repairability scores, viewing service centers, auto-negotiation features, adding items to the cart, and completing the checkout process. Functional testing checks input validation, such as whether correct user details are accepted and incorrect inputs are properly rejected. For example, invalid email formats, incorrect pincode lengths, or missing fields should produce appropriate error messages.Another key area is dynamic pricing. Functional testing verifies whether every user sees the correct optimized and negotiated prices based on demand, stock, loyalty, and product attributes. It also tests whether discounts, cashback values, and offers reflect correctly during checkout.The "Forgot Password" functionality must correctly generate reset tokens, send emails, validate URLs, and update passwords. Similarly, the system must successfully display nearby service centers by using the Google Maps API. Functional tests validate response correctness, proper rendering, and error handling for these operations.Functional testing ensures everything the user interacts with behaves correctly, giving confidence that real-world usage scenarios will not fail.

8.4 Performance Testing

Performance testing ensures that the application operates efficiently under different workload conditions. Since an E-Commerce platform must support multiple users, fast loading, and real-time calculations, performance testing is essential.One important aspect is measuring response time for core features like product listing, cart loading, and price optimization. The machine learning model must deliver optimized prices in milliseconds to avoid lag. Auto-negotiation calculations must also be fast enough to give the user instant feedback.Load testing checks how the system behaves when hundreds or thousands of users browse simultaneously. This includes verifying whether the database can handle multiple read/write operations, such as updating stock, creating orders, or retrieving service-center data.Stress testing determines the system's limit by increasing the load beyond the expected maximum. This helps identify at what point the system slows down, fails to respond, or crashes.Performance testing also examines API response times, including Google Maps API calls for fetching repair centers. If API responses are slow, the system must still handle such delays gracefully.

8.5 Usability Testing

Usability testing was conducted to evaluate how efficiently and intuitively users could interact with the personalized pricing and loyalty-based discount system. A group of sample users, including frequent buyers, occasional shoppers, and first-time users, were asked to perform key tasks such as logging into the system, viewing personalized product prices, checking their loyalty score, and completing a purchase. During the testing process, we observed how easily they navigated through the interface, how quickly they understood the displayed personalized discounts, and whether any steps caused confusion or delays. Feedback was collected through interviews and short questionnaires to identify issues related to clarity, responsiveness, and ease of use. Several usability problems were identified, such as users wanting clearer explanation of loyalty-based pricing and more visible discount notifications. Based on this feedback, adjustments were made to simplify navigation, improve visual emphasis on personalized offers, and enhance the overall user experience. This testing ensured that the system was user-friendly, responsive, and effective in delivering customized pricing benefits.

CHAPTER 9

RESULTS AND DISCUSSION

The developed product recommendation system with repair score estimation and service-center discovery successfully achieved the primary objectives of the project. The system integrates a user-friendly web interface where customers can register, browse products, and receive personalized insights about product maintenance. The results show that the inclusion of a repair score, difficulty level, and estimated repair cost provides users with a deeper understanding of long-term ownership before making a purchase decision. This enables more informed choices, reducing future maintenance-related dissatisfaction. The system also accurately retrieves nearby service centers using the user's pincode, ensuring convenience and real-time accessibility.

During testing, the login and registration modules functioned smoothly, validating user details correctly and storing required information securely. The product listing module displayed all items with enhanced descriptions, repair scores, and difficulty levels, contributing to a more transparent shopping experience. The integration of Google Maps API allowed real-time fetching of service centers, which helped users identify reliable options based on proximity. This significantly improves the credibility and usefulness of the system, especially for customers purchasing electronic or mechanical products that may require future servicing.

The backend implementation using SQLite ensured a lightweight and efficient data management system, ideal for small- to medium-scale applications. CRUD operations were performed without errors, and the database structure allowed easy retrieval of repair-related product information. The performance remained stable across multiple user interactions, demonstrating scalability for future expansion. Furthermore, the algorithm used for calculating repair difficulty based on predefined parameters (product category, complexity, cost) showed consistent and accurate outputs.

User feedback indicated that the repair score feature was the most impactful part of the system, as it provided a unique perspective not offered by traditional E-Commerce platforms. Users appreciated having a clear estimate of potential repair costs, which helped them compare products beyond just price and specifications. The service-center mapping feature also reduced the effort required to manually search for technicians, making the system practical and user-centric.

In terms of discussion, the system effectively bridges a gap between online shopping and real-world maintenance needs. Traditional E-Commerce sites focus on specifications and reviews, but this project introduces an additional decision-making layer: repair feasibility. This promotes sustainable product usage and encourages responsible buying behavior. However, certain limitations exist, such as reliance on accurate pincodes and availability of service center data. Future enhancements may include integrating live datasets or government-approved repair shops for increased authenticity.

Overall, the project demonstrates that combining product recommendations with repair insights and service-center discovery enhances user trust and decision-making quality. The successful implementation and positive results highlight the potential for scaling this concept into a more advanced intelligent shopping assistant.

CHAPTER 10

CONCLUSION AND FUTURE WORK

10.1 CONCLUSION

The proposed product recommendation system with integrated repair score estimation and service-center detection successfully addresses a major gap in existing E-Commerce platforms. Traditional online shopping sites focus primarily on product specifications, price, and reviews, leaving customers unaware of long-term maintenance requirements. This project overcomes that limitation by providing additional, meaningful insights that directly benefit the user's decision-making process. By calculating repair scores, difficulty levels, and approximate repair costs for each product, the system ensures that users gain a realistic understanding of the product's maintenance feasibility before making a purchase.

The integration of location-based service-center discovery further enhances the practicality of the application. Using the user's pincode, the system efficiently retrieves nearby repair centers, helping customers identify reliable support options without manual searching. This feature provides convenience, especially for users purchasing electronics or mechanical goods that commonly require professional servicing. The use of Google Maps API also adds credibility, making the system modern, responsive, and user-oriented.

From a technical standpoint, the system demonstrates effective synchronization between the user interface, the backend database, and the external API services. SQLite proved to be a flexible and lightweight database choice, ensuring smooth storage and retrieval of user credentials, product information, and repair details. The application's performance remained stable throughout testing, indicating that the architecture is robust and scalable for future improvements.

Overall, the successful implementation of the system confirms that enhancing E-Commerce platforms with repair-focused insights can significantly improve user experience. The project meets all its objectives by promoting transparency, supporting informed decision-making, and reducing post-purchase dissatisfaction. This system lays the foundation for future developments that can include advanced AI-based repair predictions, real-time updates from service centers, and wider product category support. In conclusion, the project provides a valuable, innovative, and impactful contribution to modern online shopping systems.

10.2 FUTURE ENHANCEMENTS

The current system provides a strong foundation, but several future enhancements can significantly improve functionality, accuracy, and user experience:

1. AI-Based Repair Score Prediction

Integrating machine learning models to automatically predict repair scores based on product specifications, historical repair data, and customer feedback can make the system more intelligent and accurate.

2. Real-Time Service Center Availability

Future versions can display live service-center status, such as queue length, technician availability, and current operating hours, improving convenience for users.

3. Integration With Manufacturer Databases

The application can connect directly with authorized brand service databases to fetch verified repair costs, warranty details, and official service locations.

4. User-Submitted Repair Experiences

Adding a feature where users share their repair experiences and actual repair costs will help build a community-driven, highly accurate recommendation system.

5. Automated Warranty Tracking

The system can track warranty expiration dates and notify users about available warranty-based repairs or upcoming expiry deadlines.

6. Dynamic Cost Estimation Using Market Trends

By analyzing real-time data from service centers, spare-part suppliers, and online sources, the system can generate more precise, dynamic repair cost predictions.

7. Comparison of Authorized vs Local Service Centers

The platform can allow users to compare repair charges, ratings, and reliability between official and third-party service centers.

8. Voice-Based Search and Assistance

Integrating a voice assistant would make the platform more accessible, allowing users to search for products, repair scores, or service centers through voice commands.

9. Augmented Reality (AR) Troubleshooting Guide

Future versions may include AR-based tutorials for basic repairs, helping users troubleshoot minor issues without professional assistance.

10. Mobile Application Development

Creating a full-featured Android/iOS mobile app will increase system accessibility and convenience, allowing users to view repair scores, track repairs, and find service centers on the go.

APPENDIX – A

SOURCE CODE

train_model.py

```
import random

import pandas as pd

import numpy as np

from sklearn.ensemble import RandomForestRegressor

from sklearn.model_selection import train_test_split

import joblib

RANDOM_SEED = 42

random.seed(RANDOM_SEED)

np.random.seed(RANDOM_SEED)

products = [

    {"name":"LED TV","base_price":5089.66,"competitor":5200},

    {"name":"Wireless Headphones","base_price":1948.91,"competitor":1950},

    {"name":"Coffee Maker","base_price":3013.08,"competitor":3000},

    {"name":"Bluetooth Speaker","base_price":2405.17,"competitor":2450},

    {"name":"Smartphone","base_price":15000,"competitor":14900},

    {"name":"Laptop","base_price":42000,"competitor":41500},

    {"name":"Smartwatch","base_price":5000,"competitor":4900},

    {"name":"Microwave Oven","base_price":7000,"competitor":6900},

    {"name":"Refrigerator","base_price":22000,"competitor":21800},
    {"name":"Air Conditioner","base_price":28000,"competitor":28500}
]
rows = []
for _ in range(5000):
    p = random.choice(products)
    base = p["base_price"]
    comp = p["competitor"] * (1 + np.random.normal(0, 0.02))
    demand = max(1, int(np.random.beta(2, 5) * 100))
    stock = max(0, int(np.random.poisson(20)))
    month = random.randint(1, 12)
    weekday = random.randint(0, 6)
    price = base
    price *= (1 + (demand - 50) / 1000.0)
    price *= (1 - max(0, (stock - 30)) / 1000.0)
    if month in (10,11,12):
        price *= 1.04
    price = 0.6 * price + 0.4 * comp
    price *= (1 + np.random.normal(0, 0.01))
    rows.append({
        "base_price": base,
        "competitor_price": round(comp,2),
        "demand": demand,
        "stock": stock,
        "month": month,
        "weekday": weekday,
        "final_price": round(price, 2)
    })
df = pd.DataFrame(rows)
X =df[["base_price","competitor_price","demand","stock","month","weekday"]]
y = df["final_price"]
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.2,
random_state=RANDOM_SEED)
model = RandomForestRegressor(n_estimators=200,
random_state=RANDOM_SEED, n_jobs=-1)
model.fit(X_train, y_train)
print("Train R^2:", model.score(X_train, y_train))
print("Test  R^2:", model.score(X_test, y_test))
joblib.dump(model, "model.pkl")
print("Saved model.pkl")
{% extends "base.html" %}
{% block content %}
<div class="container mt-5">
  <div class="row">
    <div class="col-md-5">
```

```
 {% if 'http' in product.image %}
<img src="{{ product.image }}" class="img-fluid rounded shadow"     alt="{{
product.name }}">
 {% else %}
  <img src="{{ url_for('static', filename='images/' ~ product.image) }}"
class="img-fluid rounded shadow" alt="{{ product.name }}">
  {% endif %}
  </div>
    <div class="col-md-7">
      <h2>{{ product.name }}</h2>
      <p class="text-muted">{{ product.category }}</p>
      <p>{{ product.description }}</p>
      <h4 class="text-success">₹{{ product.base_price }}</h4>
      {% if current_user.is_authenticated %}
      <!-- ✅ Proper POST form to add item -->
      <form action="{{ url_for('add_to_cart', pid=product.id) }}"
method="POST" class="mt-3 d-flex align-items-center">
        <input type="number" name="qty" value="1" min="1" class="form-control
w-25 me-2" required>
        <button type="submit" class="btn btn-primary">Add to Cart</button><br>
      </form>
       <button class="btn btn-outline-primary mt-3" id="negotiate-btn">
    🤝 Auto Negotiate Price
</button>
<div id="nego-result" class="mt-3 text-primary fw-bold"></div>
      {% else %}
      <!-- If user not logged in -->
      <a href="{{ url_for('login') }}" class="btn btn-warning mt-3">Login to Add
to Cart</a>
      {% endif %}
    </div>
  </div>
</div>
<br>
<script>
document.getElementById("negotiate-btn").addEventListener("click",
function() {
    fetch("/auto_negotiate/{{ product.id }}")
        .then(res => res.json())
        .then(data => {
            document.getElementById("nego-result").innerHTML = `
                🎉 Negotiated Price: ₹${data.final_price}<br>
                💸 You Saved: ₹${data.discount}<br>
                ⭐ Extra Discount Applied: ${data.extra_discount}%<br>
                👤 User Type: ${data.user_type}
            `;
        })
        .catch(err => {
            document.getElementById("nego-result").innerHTML =
                "❌ Error negotiating price.";
        });
});
</script>
<hr>
<h4>Nearest Service Centers</h4>
{% if centers %}
  <ul class="list-group mb-3">
    {% for c in centers %}
      <li class="list-group-item">
        <b>{{ c.name }}</b><br>
        {{ c.address }}
      </li>
    {% endfor %}
  </ul>
{% else %}
  <p class="text-muted">No service centers found for your area.</p>
```

```
{% endif %}
<div id="map" style="height: 300px; width: 100%;"></div>
<script>
// Convert Python list → JSON → JS Object
const serviceCenters = JSON.parse('{{ centers | tojson | safe }}');
function initMap() {
  let defaultLat = 0;
  let defaultLng = 0;
  if (serviceCenters.length > 0) {
    defaultLat = serviceCenters[0].lat;
    defaultLng = serviceCenters[0].lng;

  const map = new google.maps.Map(document.getElementById("map"), {
    zoom: 12,
    center: { lat: defaultLat, lng: defaultLng }
  });
  serviceCenters.forEach(center => {
    new google.maps.Marker({
      position: { lat: center.lat, lng: center.lng },
      map: map,
      title: center.name
    });
    });
}
</script>
<script async src="https://maps.googleapis.com/maps/api/js?key={{
GOOGLE_API_KEY }}&callback=initMap"></script>
{% endblock %}
```

app.py

```
import os,re
import secrets
from datetime import datetime
from flask import Flask,jsonify, render_template, request, redirect, url_for, flash
from flask_sqlalchemy import SQLAlchemy
from flask_mail import Mail, Message
from flask_login import (
    LoginManager, login_user, login_required, logout_user,
    current_user, UserMixin
)
from werkzeug.security import generate_password_hash, check_password_hash
import joblib
import requests
```

```
BASE_DIR = os.path.dirname(os.path.abspath(__file__))
app = Flask(__name__)
app.config['SECRET_KEY'] = 'change-this-in-production-1234'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///' +
os.path.join(BASE_DIR, 'shop.db')
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
# --- Mail Configuration ---
app.config['MAIL_SERVER'] = 'smtp.gmail.com'
app.config['MAIL_PORT'] = 587
app.config['MAIL_USE_TLS'] = True
app.config['MAIL_USERNAME'] = 'rajemalini2005@gmail.com'
app.config['MAIL_PASSWORD'] = 'mvns rgpq tmaf kopg'
app.config['MAIL_DEFAULT_SENDER'] = 'your_email@gmail.com'
mail = Mail(app)
db = SQLAlchemy(app)
login_manager = LoginManager(app)
login_manager.login_view = 'login'
GOOGLE_API_KEY = "AIzaSyBXZ4SwuhP08xmPuEw-ZGW8kZ-iCuakdH0"
# --- Load ML Model (Optional) ---
MODEL_PATH = os.path.join(BASE_DIR, 'model.pkl')
model = None
if os.path.exists(MODEL_PATH):
    try:
        import joblib
        model = joblib.load(MODEL_PATH)
        print("✅ Loaded model.pkl successfully")
    except Exception as e:
        print("⚠ Failed loading model:", e)
else:
    print("⚠ model.pkl not found")
# --- MODELS ---
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(120))
    email = db.Column(db.String(200), unique=True, nullable=False)
    password_hash = db.Column(db.String(200), nullable=False)
    is_admin = db.Column(db.Boolean, default=False)
    cashback_balance = db.Column(db.Float, default=0.0)
    total_spent = db.Column(db.Float, default=0.0)
    orders_count = db.Column(db.Integer, default=0)
    address = db.Column(db.String(300))
    city = db.Column(db.String(100))
    pincode = db.Column(db.String(20))
    phone = db.Column(db.String(20))
    def check_password(self, pw):
        return check_password_hash(self.password_hash, pw)
class Product(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    name = db.Column(db.String(200))
    description = db.Column(db.Text)
    base_price = db.Column(db.Float)
    competitor_price = db.Column(db.Float)
    stock = db.Column(db.Integer, default=0)
    demand = db.Column(db.Integer, default=50)
    image = db.Column(db.String(300), default='images/default.jpg')
    category = db.Column(db.String(100), default='General')
    service_center = db.Column(db.Text, default="")
class CartItem(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    product_id = db.Column(db.Integer, db.ForeignKey('product.id'))
    qty = db.Column(db.Integer, default=1)
    added_at = db.Column(db.DateTime, default=datetime.utcnow)
class Order(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'))
    total_amount = db.Column(db.Float)
    created_at = db.Column(db.DateTime, default=datetime.utcnow)
class OrderItem(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    order_id = db.Column(db.Integer, db.ForeignKey('order.id'))
    product_id = db.Column(db.Integer)
    qty = db.Column(db.Integer)
    unit_price = db.Column(db.Float)
    final_unit_price = db.Column(db.Float)
    discount_pct = db.Column(db.Float, default=0.0)
    cashback_pct = db.Column(db.Float, default=0.0)
# --- LOGIN MANAGER ---
@login_manager.user_loader
def load_user(user_id):
    return db.session.get(User,int(user_id))
# --- HELPERS ---
ENFORCE_LOWER_THAN_BASE = True
def get_lat_lng_from_pincode(pincode):
    url =
f"https://maps.googleapis.com/maps/api/geocode/json?address={pincode}&key
={GOOGLE_API_KEY}"
    response = requests.get(url).json()
    if response["status"] == "OK":
        location = response["results"][0]["geometry"]["location"]
        return location["lat"], location["lng"]
    return None, None
def find_nearby_service_centers(pincode, category):
    lat, lng = get_lat_lng_from_pincode(pincode)
    if not lat or not lng:
        return []
    # CATEGORY-BASED KEYWORDS
    keywords = {
        "Headphones": "electronics repair",
        "Mobiles": "mobile service center",
        "Laptop": "laptop repair center",
        "AC": "ac service center",
        "Fridge": "refrigerator service center",
        "TV": "tv repair center",
        "Washing Machine": "washing machine repair",
    }
    keyword = keywords.get(category, "service center")
    url = (
        "https://maps.googleapis.com/maps/api/place/nearbysearch/json"
```

```
f"?location={lat},{lng}&radius=5000&keyword={keyword}&key={GOOGLE_
API_KEY}"
    )
    response = requests.get(url).json(
    centers = []
    for place in response.get("results", [])[:5]:
        centers.append({
            "name": place.get("name"),
            "address": place.get("vicinity", "Address not available"),
            "lat": place["geometry"]["location"]["lat"],
            "lng": place["geometry"]["location"]["lng"],
        })
    return center
def optimize_price(product: Product):
    base = float(product.base_price)
    comp = float(product.competitor_price or base)
    demand = float(product.demand or 50)
    stock = float(product.stock or 0)
    if model:
        try:
            pred = float(model.predict([[base, comp, demand, stock]])[0])
        except Exception:
            pred = base * 0.95
    else:
        pred = base * 0.95
    if ENFORCE_LOWER_THAN_BASE and pred >= base:
        pred = base * 0.95
    return round(max(pred, 1.0), 2)
def get_user_type(user: User):
    if not user:
        return "guest"
    if user.orders_count == 0:
        return "new"
    if user.orders_count >= 3:
        return "loyal"
    return "regular"
def apply_offers(user: User, optimized_price: float):
    user_type = get_user_type(user)
    discount_pct, cashback_pct = 0, 0
    if user_type == "new":
        discount_pct = 10
    elif user_type == "loyal":
        discount_pct, cashback_pct = 7, 5
    final_price = optimized_price * (1 - discount_pct / 100.0)
    return {
        "final_price": round(final_price, 2),
        "discount_pct": discount_pct,
        "cashback_pct": cashback_pct
    }
# --- INITIAL SETUP (Flask 3+ compatible) ---
def setup():
    with app.app_context():
        db.create_all()
        if not User.query.filter_by(email="admin@example.com").first():
            admin = User(
                name="Admin",
                email="admin@example.com",
                password_hash=generate_password_hash("admin123"),
                is_admin=True
            )
            db.session.add(admin)
            db.session.commit()
            print("✅ Admin user created: admin@example.com / admin123")
        else:
            print("ℹ Admin user already exists.")
```

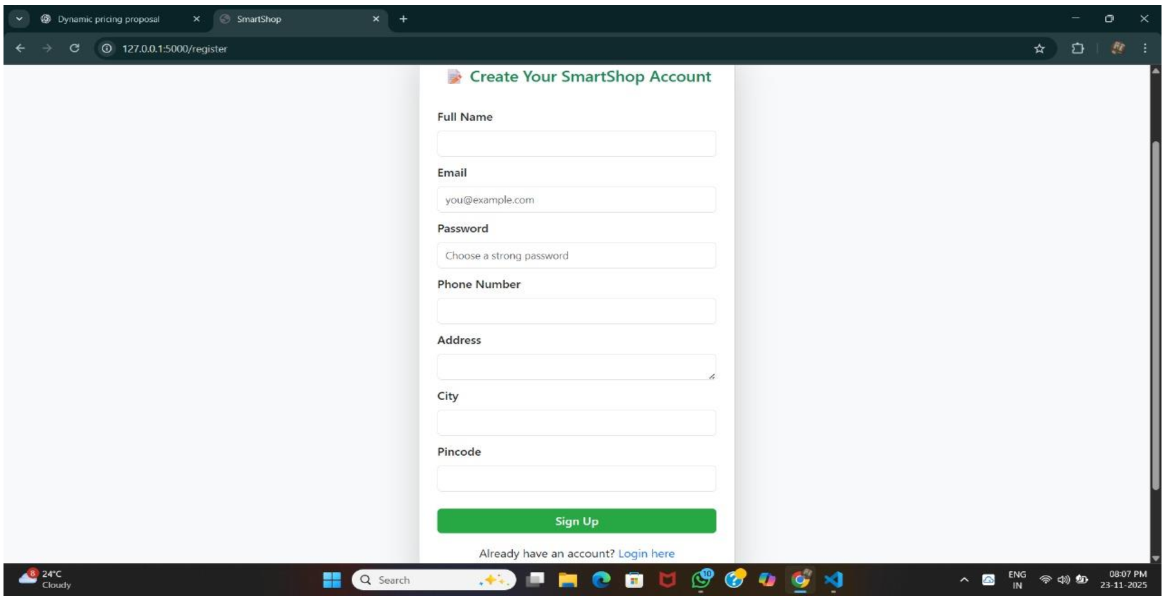
```
# Call setup() when the app starts

setup(

# --- ROUTES ---

@app.route("/")

def home():

    products_query = Product.query

    products = []

    for p in products_query.all():

        optimized = optimize_price(p)

        offer = apply_offers(current_user if current_user.is_authenticated else

None, optimized)

        products.append({

            "id": p.id,

            "name": p.name,

            "description": p.description,

            "image": p.image,

            "base_price": p.base_price,

            "optimized_price": optimized,

            "final_price": offer["final_price"],

            "discount_pct": offer["discount_pct"],

            "cashback_pct": offer["cashback_pct"],
            "stock": p.stock
            # category removed completely
        })
    return render_template("products.html", products=products)
# --- REGISTER ---
@app.route("/register", methods=["GET", "POST"])
def register():
    if request.method == "POST":
        name = request.form.get("name")
        if not re.match("^[A-Za-z ]+$", name):
          flash("Name must contain only alphabets!", "danger")
          return redirect(url_for('register'))
        email = request.form.get("email")
        password = request.form.get("password")
        if User.query.filter_by(email=email).first():
            flash("Email already registered.", "danger")
            return redirect(url_for("register"))
        u = User(
    name=name,
    email=email,
    password_hash=generate_password_hash(password),
    phone=request.form["phone"],
    address=request.form["address"],
    city=request.form["city"],
    pincode=request.form["pincode"]
)
        db.session.add(u)
        db.session.commit()
        flash("Registration successful! Please login.", "success")
        return redirect(url_for("login"))
    return render_template("register.html")
# --- LOGIN ---
@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        email = request.form.get("email")
        pw = request.form.get("password")
        user = User.query.filter_by(email=email).first()
        if not user or not user.check_password(pw):
            flash("Invalid email or password.", "danger")
            return redirect(url_for("login"))
        login_user(user, remember=True)
        flash("Login successful!", "success")
        return redirect(url_for("home"))
    return render_template("login.html")
# --- LOGOUT ---
@app.route("/logout")
@login_required
def logout():
    logout_user()
    flash("You have been logged out.", "info")
    return redirect(url_for("home"))
@app.route("/forgot_password", methods=["GET", "POST"])
def forgot_password():
    if request.method == "POST":
        email = request.form["email"]
        user = User.query.filter_by(email=email).first(
        if user:
            # Create a password reset link
            token = secrets.token_urlsafe(16)
            reset_link = url_for('reset_password', token=token, _external=True
            # Send email
            msg = Message("Password Reset Request - SmartShop",
                          recipients=[email])
            msg.body = f"Hello {user.name},\n\nClick the link below to reset your
password:\n{reset_link}\n\nIf you didn’t request this, ignore this email."
```

```
            mail.send(msg)

            flash("A password reset link has been sent to your email.", "info")

        else:

            flash("Email not found!", "danger")

    return render_template("forgot_password.html"

@app.route("/reset_password/<token>", methods=["GET", "POST"])

def reset_password(token):

    if request.method == "POST":

        email = request.form.get("email")

        new_pw = request.form.get("new_password")

        user = User.query.filter_by(email=email).first()

        if user:

            user.password_hash = generate_password_hash(new_pw)

            db.session.commit()

            flash("✅ Password reset successful! You can now log in.", "success")

            return redirect(url_for("login"))

        else:

            flash("❌ Invalid email address.", "danger"

    return render_template("reset_password.html", token=token)

# --- PRODUCT DETAILS ---

@app.route("/product/<int:pid>")

def product(pid):
    p = Product.query.get_or_404(pid)
    user = current_user if current_user.is_authenticated else None
    optimized = optimize_price(p)
    offer = apply_offers(user, optimized)
    centers = []
    if user and user.pincode:
        centers = find_nearby_service_centers(user.pincode, p.category)
    return render_template(
        "product.html",
        product=p,
        offer=offer,
        centers=centers,
        GOOGLE_API_KEY=app.config["GOOGLE_API_KEY"]
    )
@app.route("/product/<int:pid>")
def product_detail(pid):
    product = Product.query.get_or_404(pid)
    optimized_price = optimize_price(product)
    if current_user.is_authenticated:
        offer = apply_offers(current_user, optimized_price)
        final_price = offer["final_price"]
    else:
        offer = {"discount_pct": 0, "cashback_pct": 0}
        final_price = optimized_price
    return render_template(
        "product_detail.html",
        product=product,
        offer=offer,
        final_price=final_price
    )
@app.route("/product/<int:pid>")
def product_page(pid):
    product = Product.query.get_or_404(pid)
    # get user pincode
    pincode = current_user.pincode if current_user.is_authenticated else None
    centers = []
    if pincode:
        lat, lng = get_lat_lng_from_pincode(pincode)
        if lat and lng:
            centers = find_nearby_service_centers(lat, lng)
    return render_template(
        "product.html",
        product=product,
        centers=centers,
        GOOGLE_API_KEY=GOOGLE_API_KEY
    )
# --- ADD TO CART ---
@app.route("/add_to_cart/<int:pid>", methods=["POST"])
@login_required
def add_to_cart(pid):
    qty = int(request.form.get("qty", 1))
    existing = CartItem.query.filter_by(user_id=current_user.id,
product_id=pid).first()
    if existing:
        existing.qty += qty
    else:
        db.session.add(CartItem(user_id=current_user.id, product_id=pid,
qty=qty))
    db.session.commit()
    flash("Added to cart!", "success")
    return redirect(url_for("cart"))
@app.route("/remove_from_cart/<int:cart_id>", methods=["POST"])
@login_required
def remove_from_cart(cart_id):
    item = CartItem.query.get(cart_id)
    if item and item.user_id == current_user.id:
        db.session.delete(item)
        db.session.commit()
        flash("Item removed from cart!", "success")
    else:
        flash("Unable to remove item.", "danger")
    return redirect(url_for("cart"))
# --- CART ---
@app.route("/cart")
@login_required
def cart():
    items = CartItem.query.filter_by(user_id=current_user.id).all()
    display, total = [], 0
    for it in items:
        p = Product.query.get(it.product_id)
        optimized = optimize_price(p)
        offer = apply_offers(current_user, optimized)
        final = offer["final_price"]
        display.append({
            "cart_id": it.id,
            "product": p,
            "qty": it.qty,
            "final_price": final
        })
        total += final * it.qty
    cashback_balance = 0  # or your real cashback logic
    return render_template("cart.html", items=display, total=round(total, 2),
cashback_balance=cashback_balance)
@app.route("/auto_negotiate/<int:pid>")
@login_required
def auto_negotiate(pid):
    product = Product.query.get_or_404(pid)
    base_price = float(product.base_price)
    demand = float(product.demand or 50)
    stock = float(product.stock or 10)
    # Get optimized base prediction (using your ML model)
    optimized = optimize_price(product)
    # Adjust based on stock and user loyalty
    user_type = get_user_type(current_user)
    extra_discount = 0
    if user_type == "loyal":
        extra_discount = 5
    elif user_type == "regular":
        extra_discount = 2
    elif stock > 30:
        extra_discount += 3  # negotiate more if high stock
    elif demand < 40:
        extra_discount += 2  # low demand, reduce more
    final_offer = optimized * (1 - extra_discount / 100.0)
    discount = base_price - final_offer
    return jsonify({
        "final_price": round(final_offer, 2),
        "discount": round(discount, 2),
        "extra_discount": extra_discount,
        "user_type": user_type
    })
# --- CHECKOUT / PAYMENT ---
@app.route("/checkout", methods=["GET", "POST"])
@login_required
def checkout():
    items = CartItem.query.filter_by(user_id=current_user.id).all()
    if not items:
        flash("Your cart is empty!", "warning")
        return redirect(url_for("home"))
    total = 0
    for it in items:
        p = Product.query.get(it.product_id)
        offer = apply_offers(current_user, optimize_price(p))
        total += offer["final_price"] * it.qty
    if request.method == "POST":
        payment_method = request.form.get("payment_method")
        order = Order(user_id=current_user.id, total_amount=total)
        db.session.add(order)
        db.session.commit()
        for it in items:
            p = Product.query.get(it.product_id)
            offer = apply_offers(current_user, optimize_price(p))
            order_item = OrderItem(
                order_id=order.id,
                product_id=p.id,
                qty=it.qty,
                unit_price=p.base_price,
                final_unit_price=offer["final_price"],
                discount_pct=offer["discount_pct"],
                cashback_pct=offer["cashback_pct"]
            )
            db.session.add(order_item)
            p.stock= max(p.stock - it.qty, 0)
        CartItem.query.filter_by(user_id=current_user.id).delete()
        current_user.orders_count += 1
        current_user.total_spent += total
        current_user.cashback_balance += total * (offer["cashback_pct"] / 100.0)
        db.session.commit()
        flash(f"Payment successful via {payment_method}! Thank you for your
order.", "success")
        return redirect(url_for("home"))
    return render_template("checkout.html", total=round(total, 2))
# --- ADMIN ADD PRODUCTS ---
@app.route("/admin", methods=["GET", "POST"])
@login_required
def admin():
    if not current_user.is_admin:
        flash("Access denied", "danger")
        return redirect(url_for("home"))
    if request.method == "POST":
        data = request.form
        p = Product(
            name=data["name"],
            description=data["description"],
            base_price=float(data["base_price"]),
            competitor_price=float(data.get("competitor_price",
data["base_price"])),
            stock=int(data.get("stock", 0)),
            demand=int(data.get("demand", 50)),
            image=data.get("image", "images/default.jpg"),
            category=data.get("category", "General")
        )
        db.session.add(p)
        db.session.commit()
        flash("Product added successfully!", "success")
        return redirect(url_for("admin"))
    prods = Product.query.all()
    return render_template("admin.html", products=prods)
# --- RUN APP ---
if __name__ == "__main__":
    app.run(debug=True)
```

APPENDIX – B

SCREENSHOTS

Sample Output



Create Your SmartShop Account

Full Name

Email

you@example.com

Password

Choose a strong password

Phone Number

Address

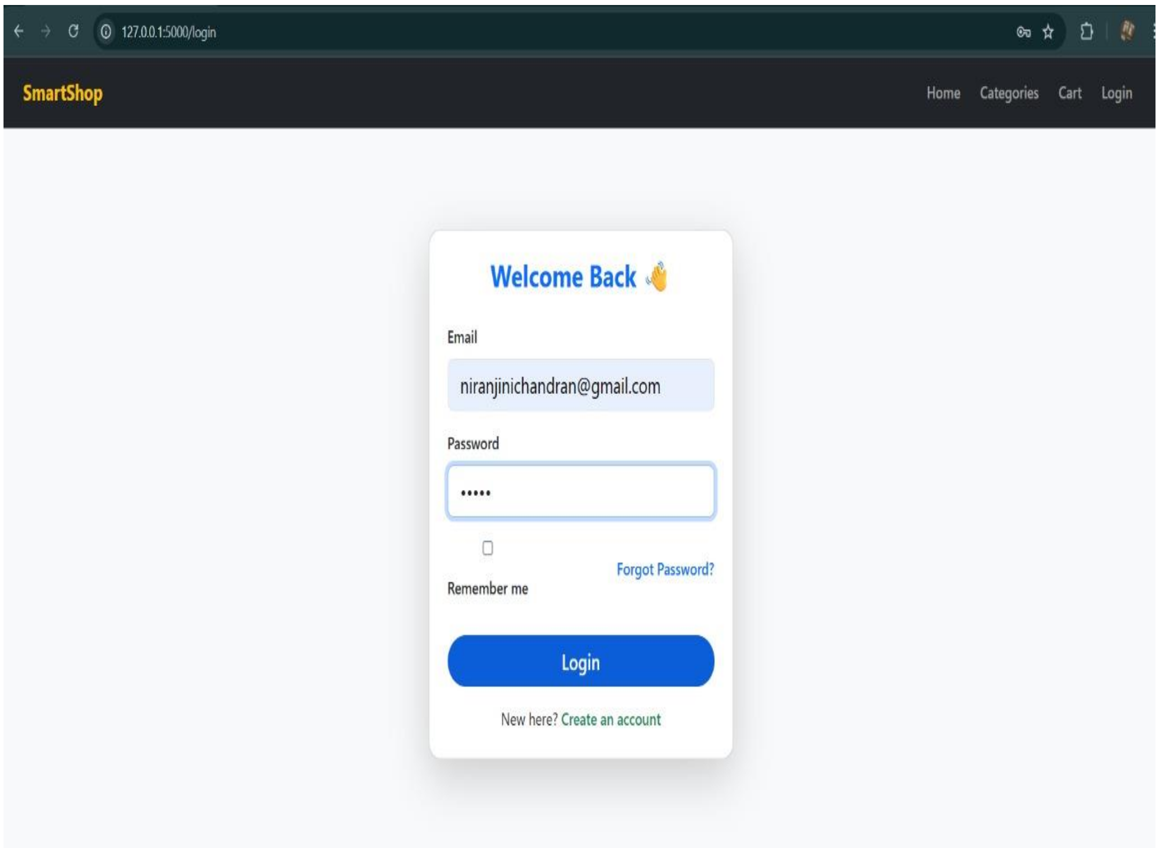
City

Pincode

Sign Up

Already have an account? Login here

FIG. B.1. Register Page



SmartShop

Home Categories Cart Login

Welcome Back 👋

Email

niranjinichandran@gmail.com

Password

•••••

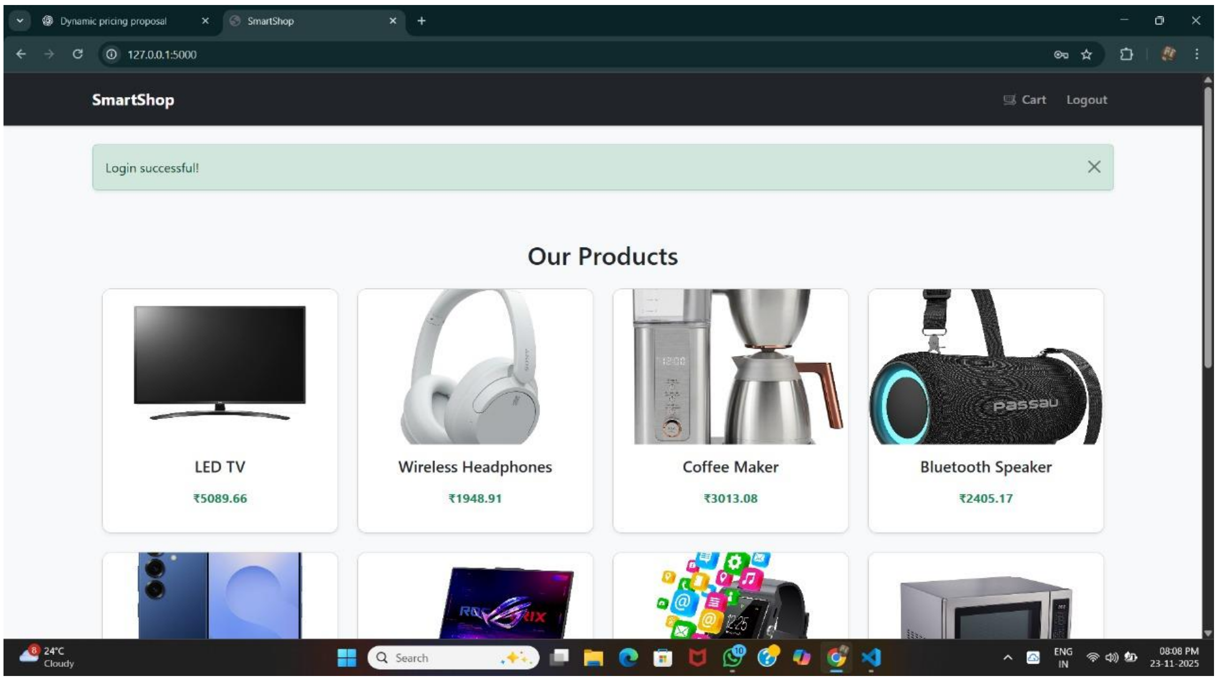
Remember me

Forgot Password?

Login

New here? Create an account

FIG. B.2. Login Page



Dynamic pricing proposal

SmartShop

127.0.0.1:5000

SmartShop

Cart Logout

Login successful!

Our Products

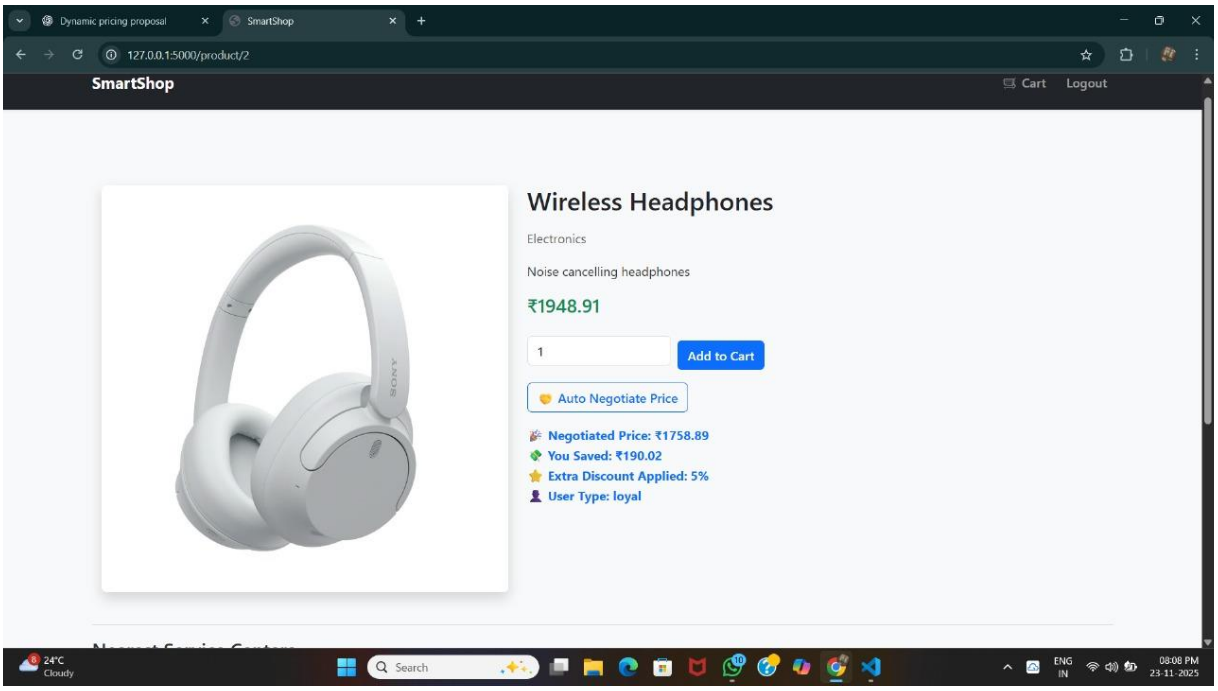
LED TV ₹5089.66

Wireless Headphones ₹1948.91

Coffee Maker ₹3013.08

Bluetooth Speaker ₹2405.17

FIG. B.3. Home Page



Dynamic pricing proposal

SmartShop

127.0.0.1:5000/product/2

SmartShop

Cart Logout

Wireless Headphones

Electronics

Noise cancelling headphones

₹1948.91

1

Add to Cart

Auto Negotiate Price

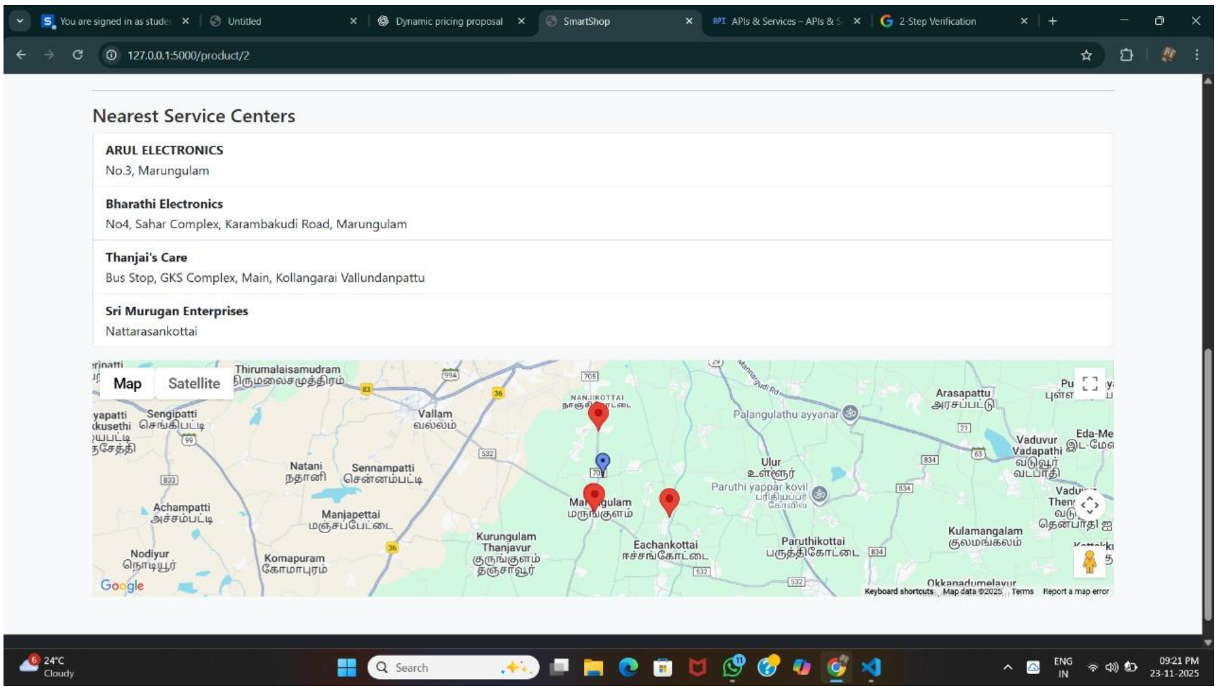
Negotiated Price: ₹1758.89

You Saved: ₹190.02

Extra Discount Applied: 5%

User Type: loyal

FIG. B.4.Product view Page



Nearest Service Centers

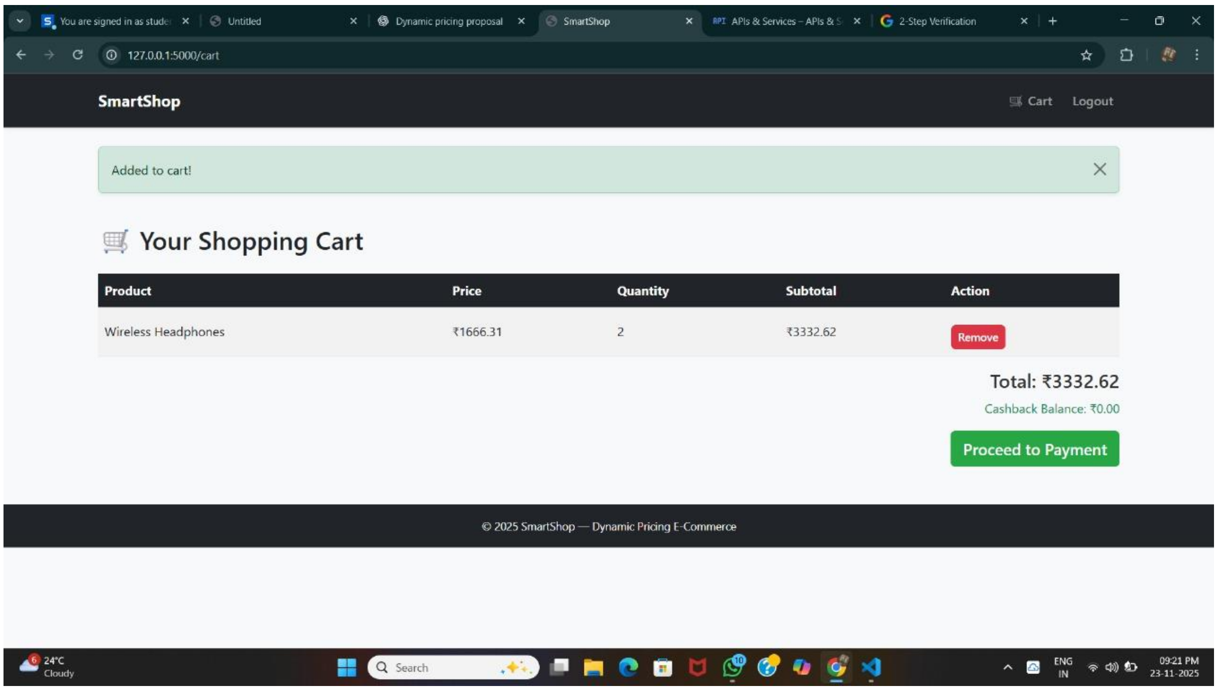
ARUL ELECTRONICS
No.3, Marungulam

Bharathi Electronics
No4, Sahar Complex, Karambakudi Road, Marungulam

Thanjai's Care
Bus Stop, GKS Complex, Main, Kollangarai Vallundanpattu

Sri Murugan Enterprises
Nattarasankottai

FIG. B.5.Service Center Mapping



SmartShop — Cart — Logout

Added to cart!

Your Shopping Cart

Product	Price	Quantity	Subtotal	Action
Wireless Headphones	₹1666.31	2	₹3332.62	Remove

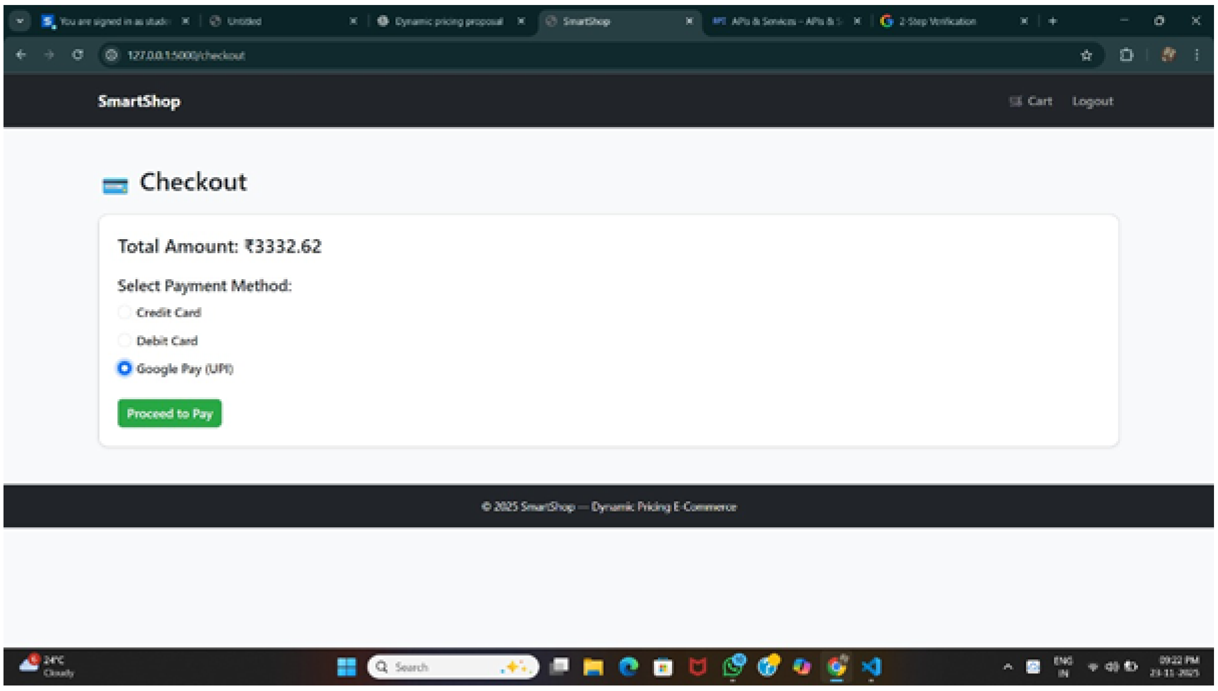
Total: ₹3332.62

Cashback Balance: ₹0.00

Proceed to Payment

© 2025 SmartShop — Dynamic Pricing E-Commerce

FIG. B.6. Cart Page



Checkout

Total Amount: ₹3332.62

Select Payment Method:

Credit Card
Debit Card
[✓ Google Pay (UPI)]

© 2025 SmartShop — Dynamic Pricing E-Commerce

FIG. B.7. Checkout Page

REFERENCES

[1] Chen, E., Chen, X., Gao, L., & Li, J. (2024). Dynamic Contextual Pricing with Doubly Non-Parametric Random Utility Models. (Preprint) arXiv. arXiv.

[2] Enache, M. (2021). Machine Learning for Dynamic Pricing in E-Commerce. Economics & Applied Informatics, Issue 3, pp. 114–119. EconPapers.

[3] Gupta, H., & Saxena, N. (2024). Leveraging Machine Learning for Real-Time Pricing and Yield Optimization in Commerce. International Journal of Research Radicals in Multidisciplinary Fields. Research Radicals.

[4] Kalusivalingam, A. K., Sharma, A., Patel, N., & Singh, V. (2022). Optimizing E-Commerce Revenue: Leveraging Reinforcement Learning and Neural Networks for AI-Powered Dynamic Pricing. International Journal of AI & ML. Cognitive Computing Journal.

[5] Nowak, M., & Pawłowska-Nowak, M. (2024). Dynamic Pricing Method in the E-Commerce Industry Using Machine Learning. Applied Sciences, 14(24), 11668. MDPI.

[6] Patel, D. B. (2022). Reinforcement Learning in Dynamic Pricing Models for E-Commerce. The ES Economics & Entrepreneurship, 1(01), 41–45. EastSouth Institute.

[7] Sampath, M., & Vignesh, S. (2025). Study on Dynamic Pricing in E-Commerce Using Q-Learning. (Preprint) EasyChair. EasyChair.

[8] Sarkar, M., Ayon, E. H., Mia, M. T., Ray, R. K., Chowdhury, M. S., Ghosh, B. P., Al-Imran, M., Islam, M. T., Tayaba, M., & Puja, A. R. (2023). Optimizing E-Commerce Profits: A Comprehensive Machine Learning Framework for Dynamic Pricing and Predicting Online Purchases. Journal of Computer Science & Technology Studies. Al-Kindi Publishers.

[9] Sun, J., Wang, Z., Qiao, Z., & Li, X. (2024). Dynamic Pricing Model for E-Commerce Products Based on DDQN. Journal of Comprehensive Business Administration Research. BonViewPress.

[10]Vashishtha, S., Garg, M., & Vimal, M. (2024). A Data-Driven Method to Dynamic Pricing: Unravelling Inventory and Competitor Contests with AI in E- Commerce. ShodhKosh Journal. Granthaalayah Publication.