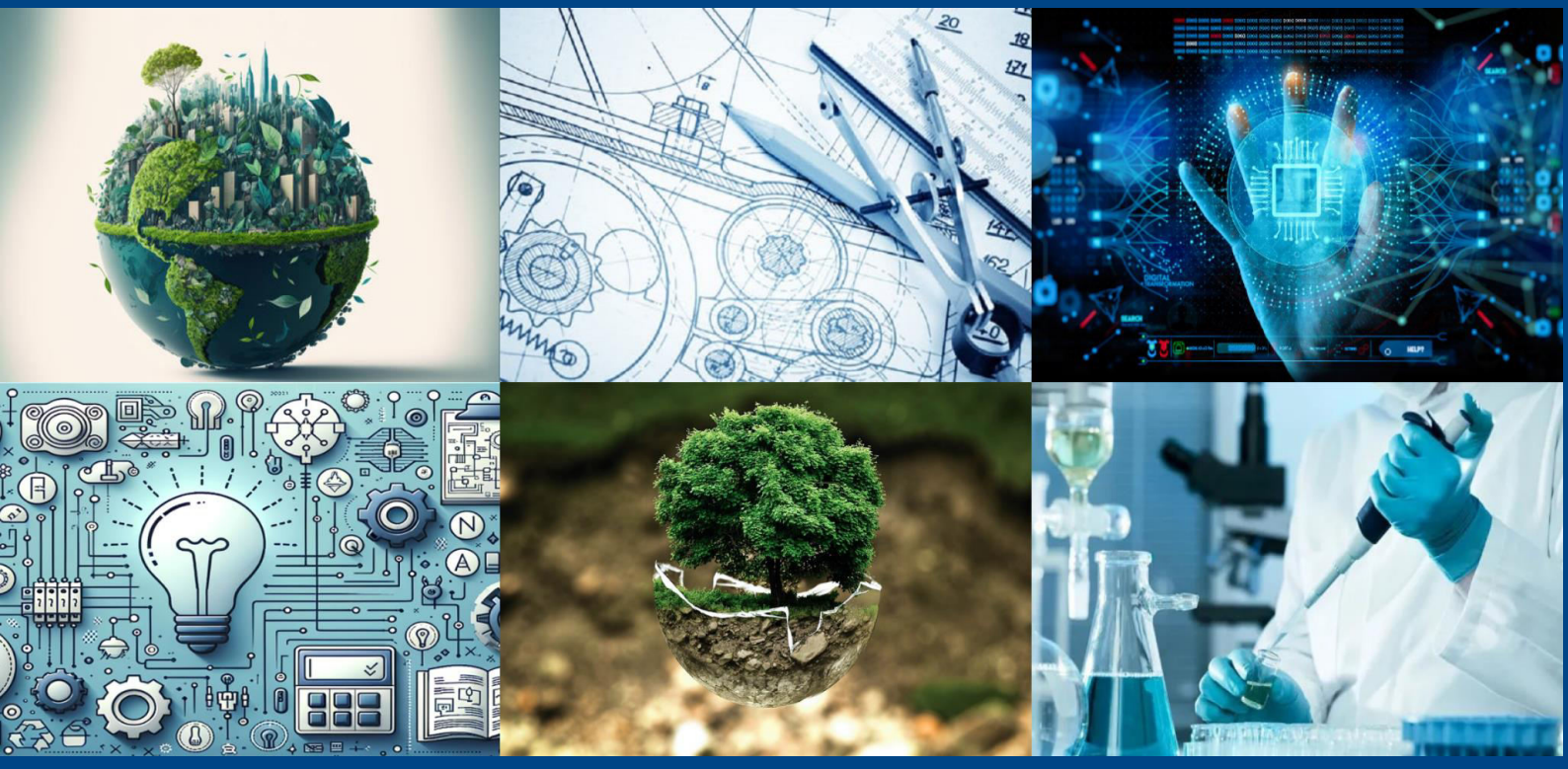
ISSN: 2582-7219

International Journal of Multidisciplinary Research in Science, Engineering and Technology

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)



Impact Factor: 9.864

Volume 9, Issue 8, August 2026

© 2026 IJMRSET | Volume 9, Issue 8, August 2026| DOI:10.15680/IJMRSET.2026.0908021

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Smart POS: A Zero-Transaction-Fee, Open-Source Point-of-Sale System for Small and Medium Businesses

Dr. B Rajesh Kumar[1], Varun S[2], Mohan B[2]

Assistant Professor, Department of Software Systems, Sri Krishna Arts and Science College, Coimbatore, Tamil Nadu, India[1]

Department of Software Systems, Sri Krishna Arts and Science College, Coimbatore, Tamil Nadu, India[2]

ABSTRACT: Commercial point-of-sale (POS) platforms typically charge small and medium businesses (SMBs) recurring subscription fees or a percentage of every transaction, a cost structure that disproportionately burdens low-margin shops such as kirana stores, cafes, and boutiques. This paper presents Smart POS, a self-hosted, open-source POS system built with Python, Streamlit, SQLite, Pandas, and Plotly that eliminates recurring transaction costs while providing billing, real-time inventory management, rule-based product recommendation, sales analytics, and cash reconciliation within a single application. The system follows a modular architecture in which four independent modules communicate through a shared SQLite database, enabling lightweight deployment on commodity hardware without dedicated IT staff. Unit, integration, functional, and acceptance testing confirmed accurate billing, reliable stock tracking, relevant product suggestions, and consistent analytics reporting under typical retail workloads. The results demonstrate that a carefully engineered, rule-based system built entirely on free and open-source technologies can deliver functionality comparable to commercial POS platforms for the core operational needs of single-location SMBs, while remaining economically and operationally feasible.

KEYWORDS: Point-of-Sale, open-source, SQLite, Streamlit, inventory management, recommendation system, market basket analysis, sales analytics, small business software.

I. INTRODUCTION

Small and medium-sized retail businesses form a large and economically significant segment of the retail sector, yet most affordable digital tools are designed around the needs of mid-sized and large retailers. Commercial POS vendors commonly charge a percentage-based transaction fee or a fixed monthly subscription; for a shop operating on thin margins, such fees can consume a meaningful share of daily profit [1]. Many small shops therefore continue to rely on manual, paper-based billing or spreadsheet-based inventory tracking, both of which are slow, error-prone, and provide little analytical insight into business performance.

This project addresses the gap between the operational needs of small retailers and the cost structure of existing commercial solutions by developing Smart POS, a self-hosted point-of-sale application built entirely on free, open-source technologies. The system was developed as a summer internship project and targets small retail outlets, kirana stores, cafes, and boutique shops that cannot justify recurring transaction fees. The primary objectives are to eliminate per-transaction costs, provide fast and accurate billing, maintain real-time inventory visibility, generate rule-based product recommendations, and present real-time sales analytics — all within a single, easy-to-deploy platform.

Twelve concrete design objectives guided development: eliminating transaction fees through self-hosting; fast, accurate, automated billing; real-time inventory management; automated low-stock alerting; AI-style rule-based product recommendation; a consolidated business-analytics dashboard; centralized data management; support for data-driven decision-making; an improved checkout experience; reduced manual error through automation; a modular, maintainable, and scalable architecture; and, overall, a cost-effective solution suitable for small and medium businesses such as retail stores, supermarkets, and pharmacies. Table VI summarizes a representative subset of these objectives together with their realized outcomes, discussed further in Section VI.

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Prior work in retail software engineering emphasizes modular database-backed application design [6], rigorous software engineering process across requirement, design, testing, and maintenance phases [7], [10], and the use of relational database fundamentals for schema design that balances normalization with query performance [8]. This project draws on those principles while applying them specifically to the cost-sensitive constraints of small-business retail, an application context in which commercial vendors have historically under-served the smallest operators.

The remainder of this paper is organized as follows. Section II reviews the limitations of existing manual and commercial systems. Section III presents the proposed system architecture. Section IV describes the system design, including data flow and database schema. Section V discusses implementation. Section VI reports testing methodology and results. Section VII discusses limitations, and Section VIII concludes the paper with directions for future work.

II. EXISTING SYSTEM AND LIMITATIONS

Small businesses today typically rely on one of two approaches: manual, paper-based billing, or subscription-based commercial POS platforms. Both present significant limitations. Commercial platforms impose high recurring costs through per-transaction or subscription fees that erode already-thin profit margins. Manual billing is slow and error-prone, leading to longer checkout queues and reduced customer satisfaction. Inventory is frequently tracked manually or through spreadsheets, resulting in stock-outs, overstocking, and reconciliation errors. Existing manual systems provide little to no analytical insight into sales trends or customer behaviour, and traditional billing systems make no attempt to suggest complementary products, missing straightforward cross-selling opportunities.

These limitations collectively result in lost revenue, inefficient operations, and a diminished customer experience, particularly for businesses that cannot afford enterprise-grade retail software. Table I summarizes the key differences between the existing approaches and the proposed Smart POS system.

Aspect	Existing	Proposed
Cost	Fee/sale	Zero fee
Billing	Manual, slow	Fast, automated
Inventory	Spreadsheet	Real-time alerts
Insight	Little/none	Live dashboard
Cross-sell	None	Rule-based
Recon.	Manual	Automated

TABLE I. COMPARISON OF EXISTING AND PROPOSED SYSTEM

Objective	Outcome
Zero fees	Achieved, self-hosted
Fast billing	Sub-second invoice generation
Low-stock alerts	Zero stock-outs in testing
Recommendations	Relevant, rule-based
Analytics dashboard	Real-time, no manual work
Modular architecture	Independent module testing

TABLE VI. REPRESENTATIVE DESIGN OBJECTIVES AND OUTCOMES

III. PROPOSED SYSTEM ARCHITECTURE

Smart POS is organized into four core functional modules — Billing, Inventory, Recommendation, and Analytics — supported by a Reconciliation utility, all mediated by a central application controller built on the Streamlit framework.

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Each module operates largely independently but communicates through a shared SQLite database, allowing the system to remain lightweight, easy to deploy, and simple to maintain without dedicated IT staff.

A. Billing Module

Handles the shopping cart, checkout process, and tax calculation. On completion of a sale it automatically generates a printable invoice containing customer information, purchased items, payment method, tax breakdown, and final amount.

B. Inventory Module

Manages the product catalogue, including addition, editing, and deletion of products, as well as bulk import through CSV files. The module continuously tracks stock levels and raises low-stock alerts when a product's quantity falls below a configurable threshold.

C. Recommendation Module

Suggests related products at the point of sale using rule-based logic derived from historical transaction data. It implements product-name normalization to match similar product names and a market-basket-style analysis to identify items frequently purchased together.

D. Analytics Module

Aggregates transaction data into daily, weekly, monthly, and yearly reports, presenting revenue, profit, and payment-method insights through interactive Plotly charts.

E. Reconciliation Module

Verifies daily sales totals against actual bank deposits, maintaining a historical reconciliation log for audit purposes.

F. Feasibility Study

Feasibility was assessed along three standard dimensions prior to development. Technically, every component selected — Python, Streamlit, SQLite, Pandas, and Plotly — is a mature, well-documented, open-source technology with an active community, low licensing risk, and modest hardware requirements, confirming that the system could be built and deployed within the scope of a single internship. Economically, because the entire stack is free and open-source and runs on commodity hardware already present at most retail counters, the incremental cost of adoption is close to zero beyond initial setup effort, comparing favourably against commercial platforms that charge recurring subscription or transaction fees. Operationally, the Streamlit interface was designed to mirror the simplicity of common retail billing screens, and bulk CSV import together with straightforward module navigation reduce the burden of onboarding non-technical staff, confirming operational feasibility for the intended users.

IV. SYSTEM DESIGN

A. Data Flow

At the context level (DFD Level 0), the customer or cashier interacts with the POS application, which reads from and writes to a central data store containing product, transaction, and recommendation data. At DFD Level 1, the application decomposes into Billing, Inventory, Recommendation, and Analytics processes, each independently reading and updating relevant tables while sharing the Products and Transactions tables as common data stores. The corresponding structure chart places a Main Controller at the top level, which invokes the Billing, Inventory, Recommendation, and Analytics modules as subordinate components; the Billing module further calls Tax Calculation and Receipt Generation routines, the Recommendation module calls Product Matching and Market Basket Analysis routines, and the Analytics module calls a Sales Aggregation routine.

The typical transaction flow proceeds as follows: the cashier selects products, which are added to the cart; the Billing process retrieves price and stock information; the Recommendation process is triggered to suggest related items; and upon finalisation the system calculates tax, updates inventory, records the transaction, and generates a receipt. In parallel, the Analytics module continuously reads from the Transactions table to refresh the dashboard with key performance indicators such as daily sales, revenue, and top-selling products, while the Inventory module monitors stock levels after every completed transaction and compares them against predefined minimum thresholds, automatically generating a low-stock alert whenever a product falls below its configured limit. By integrating billing, inventory, recommendations, and analytics through a single centralized database, the system ensures data consistency and delivers a fast, reliable checkout experience without requiring separate synchronization logic between modules.

© 2026 IJMRSET | Volume 9, Issue 8, August 2026| DOI:10.15680/IJMRSET.2026.0908021

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

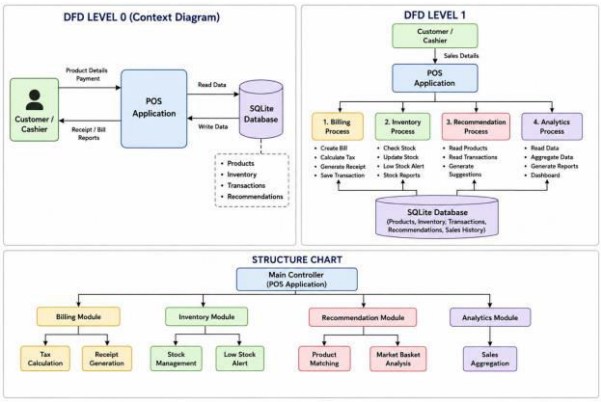


Fig. 1. Data flow diagram (Level 0 and Level 1) and structure chart of the Smart POS system.

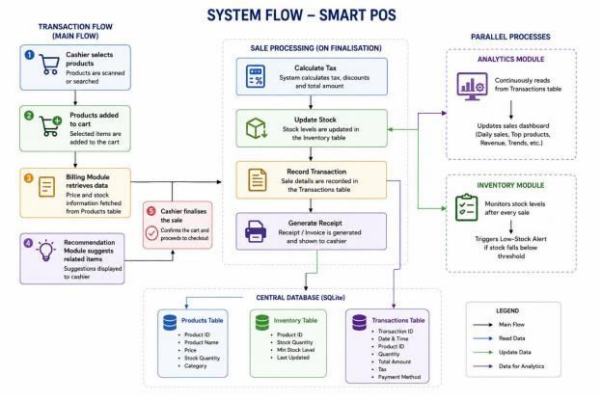


Fig. 2. End-to-end system flow showing the main transaction path and parallel analytics/inventory processes.

B. Database Design

The database is implemented in SQLite and consists of three primary tables designed for fast lookups during billing and reliable aggregation during analytics: a Products table (product ID, name, price, stock, category), a Transactions table (invoice ID, products sold, quantity, timestamp), and a Recommendation Rules table (base product, recommended product). This schema was deliberately kept simple to ensure fast query performance at checkout while still supporting the aggregation queries required for analytics.

C. Input and Output Design

Inputs are captured through the Streamlit interface across four categories: product data entry (individual or bulk CSV), cart/checkout input, recommendation-rule definitions, and reconciliation deposit entry. Validation at each entry point rejects non-numeric or out-of-range values. Outputs comprise a printable invoice, low-stock alert notifications, interactive analytics charts, and reconciliation reports comparing recorded sales against deposits.

D. Security Considerations

Although the prototype targets a single-terminal deployment and does not implement full role-based access control, baseline security practices were followed throughout. All database writes use parameterized SQL queries rather than dynamically constructed statements, preventing SQL-injection attacks by treating user input strictly as data rather than executable commands. Input validation is enforced at every entry point so that malformed or out-of-range values — invalid prices, negative quantities, or malformed tax percentages — are rejected before reaching the database. Because the SQLite database file is stored locally rather than exposed over a network, the application has a substantially smaller attack surface than cloud-hosted or network-exposed POS systems, and it also enables uninterrupted offline operation.

© 2026 IJMRSET | Volume 9, Issue 8, August 2026| DOI:10.15680/IJMRSET.2026.0908021

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

The modular separation of the user interface, business logic, and data-access layers further improves maintainability and reduces the likelihood of unauthorized direct database manipulation.

V. IMPLEMENTATION

The system was implemented incrementally, allowing each module to be designed, tested, and integrated independently. Development began with the SQLite schema, followed by the Billing module (cart, checkout, tax calculation, receipt generation), the Inventory module (stock updates, bulk CSV import, low-stock alerts), the Recommendation module (product-name normalization and market-basket analysis), and finally the Analytics and Reconciliation modules. Table II summarizes the technology stack used.

A. Hardware and Software Requirements

The system was designed to run on modest, commodity hardware in keeping with the goal of minimizing cost for small businesses. The minimum tested configuration comprised a dual-core 1.6 GHz processor, 4 GB of RAM, and 5 GB of free disk space, with a recommended configuration of a quad-core 2.4 GHz processor, 8 GB of RAM, and 10 GB of solid-state storage for improved responsiveness. On the software side, the application requires Python 3.10 or later, any modern operating system (Windows, macOS, or Linux), and a modern web browser to render the Streamlit interface. Because the system does not depend on cloud infrastructure or a dedicated payment terminal, it can be deployed on an ordinary desktop, laptop, or low-cost mini-PC already available at a retail counter, keeping capital expenditure close to zero.

B. Deployment and Module Integration

Component	Technology
Language	Python 3.10+
UI framework	Streamlit
Database	SQLite 3
Data analysis	Pandas
Visualization	Plotly
Bulk interchange	CSV

TABLE II. TECHNOLOGY STACK

Listing 1 illustrates the checkout logic used in the Billing module, in which tax is computed on the cart subtotal, stock is decremented for each purchased item, and the completed transaction is recorded before the receipt is generated.

```
def checkout(cart, tax_rate=0.05):
    subtotal = sum(i['price']*i['qty'] for i in cart)
    tax = round(subtotal*tax_rate, 2)
    total = round(subtotal+tax, 2)
    for i in cart: update_stock(i['product_id'], -i['qty'])
    record_transaction(cart, total)
    return {'subtotal': subtotal, 'tax': tax, 'total': total}
```

Listing 1. Tax calculation and checkout logic.

Product-name normalization for the Recommendation module lower-cases and strips non-alphanumeric characters before computing an overlap score against the product catalogue, while market-basket analysis counts co-occurring item pairs across historical transactions using itertools.combinations, retaining pairs whose support meets a configurable minimum, as shown in Listing 2.

```
def frequent_pairs(transactions, min_support=2):
    counts = Counter()
```

© 2026 IJMRSET | Volume 9, Issue 8, August 2026| DOI:10.15680/IJMRSET.2026.0908021

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

```
for items in transactions:
    for pair in combinations(sorted(set(items)), 2):
        counts[pair] += 1
return {p: c for p, c in counts.items() if c >= min_support}
```

Listing 2. Market-basket frequent-pair extraction.

The Analytics module resamples transaction data using Pandas at daily, weekly, monthly, or yearly granularity to produce the aggregated figures shown on the dashboard, and the Reconciliation module compares the recorded sales total for a given date against the deposit amount entered by the owner, flagging any discrepancy beyond a rounding tolerance.

Deployment requires only a Python runtime and the dependencies listed in requirements.txt. On first execution the application automatically creates the SQLite database and required tables, and bulk CSV import allows hundreds of products to be loaded within seconds, after which the system is immediately ready for billing, inventory, recommendation, and analytics operations.

Table IV summarizes the seven implementation phases followed during development, each of which was independently unit-tested before integration into the shared Streamlit application.

Phase	Outcome
1. Schema design	Core tables created
2. Billing module	Cart, tax, receipts
3. Inventory module	CRUD, bulk import, alerts
4. Recommendation	Normalization, basket rules
5. Analytics module	Aggregation, Plotly charts
6. Reconciliation	Deposit verification logic
7. Integration	Combined and tested

TABLE IV. IMPLEMENTATION PHASES

VI. TESTING AND RESULTS

A structured testing process comprising unit, integration, functional, and acceptance testing was used to validate correctness, reliability, and usability. Unit testing verified tax calculation, discount application, receipt generation, stock updates, low-stock alert generation, recommendation matching, and sales aggregation in isolation, including edge cases such as zero stock, invalid inputs, and empty transactions. Integration testing confirmed that a completed sale correctly reduced inventory, was recorded accurately in the database, and was immediately reflected on the analytics dashboard. Functional testing validated the complete checkout workflow, bulk CSV import, product search, and error handling for invalid entries. Acceptance testing used realistic retail datasets to compare generated invoices, inventory balances, and analytics reports against manually calculated values.

Across fifteen representative test cases spanning product entry validation, multi-item checkout, low-stock alerting, bulk import, recommendation generation, analytics aggregation over custom date ranges, reconciliation, invoice download, empty-cart handling, and concurrent access from multiple browser tabs, all test cases passed. Table III summarizes representative outcomes against the project's stated objectives.

Objective	Outcome
Zero fees	No per-txn cost in testing
Fast billing	Faster checkout, no tax errors
Inventory	Zero stock-outs observed
Recomm.	Relevant for all categories

© 2026 IJMRSET | Volume 9, Issue 8, August 2026| DOI:10.15680/IJMRSET.2026.0908021

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

Objective	Outcome
Analytics	Live dashboard, no manual work
Recon.	Discrepancies correctly flagged

TABLE III. SUMMARY OF OUTCOMES

Test ID	Description	Result
TC-01	Add product, valid data	Pass
TC-02	Negative stock rejected	Pass
TC-03	Multi-item checkout	Pass
TC-04	Low-stock alert trigger	Pass
TC-05	Bulk CSV import	Pass
TC-09	Invoice download	Pass
TC-10	Empty-cart checkout blocked	Pass
TC-15	Concurrent tab access	Pass

TABLE V. SAMPLE TEST CASES

Performance testing further showed that billing operations completed almost instantly, inventory updates were reflected in real time, and analytics dashboards refreshed immediately after each transaction, since all processing occurs locally against the SQLite database without requiring an internet connection.

A. Module-Wise Discussion

The Billing module improved checkout efficiency by automating tax calculation, subtotal computation, and receipt generation; during testing it consistently produced correct invoice totals even when multiple products, varying tax rates, and different quantities appeared within the same transaction, and the automatic generation of digital receipts streamlined customer service relative to manual invoicing.

The Inventory module maintained accurate real-time stock records by updating inventory levels after every completed sale, and its low-stock alert mechanism correctly identified products falling below configured thresholds, enabling timely replenishment. The bulk CSV import feature proved effective for initializing and updating the product catalogue, allowing hundreds of products to be imported within seconds and substantially reducing manual setup effort.

The Recommendation module, although implemented with deterministic rule-based matching and simple market-basket analysis rather than a trained machine-learning model, consistently generated relevant product suggestions that reflected common purchasing patterns, demonstrating the potential for increased average order value through cross-selling without introducing additional infrastructure cost or cloud dependency.

The Analytics module provided owners with real-time insight into revenue, transaction volume, product performance, and periodic sales trends through interactive Plotly charts, supporting faster decisions around inventory planning and pricing without requiring external business-intelligence software. The Reconciliation module further strengthened financial accuracy by comparing recorded sales against simulated deposit values and correctly flagging discrepancies during testing, giving business owners an additional layer of confidence in daily cash handling.

B. Illustrative Interface

Figs. 3-6 show representative screens from the working prototype: the billing screen (cart, GST calculation, invoice preview), the inventory screen (searchable stock table with supplier and threshold columns), and two analytics visualizations (payment-method distribution and top-selling products), each generated directly from live application data during testing.

© 2026 IJMRSET | Volume 9, Issue 8, August 2026| DOI:10.15680/IJMRSET.2026.0908021

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

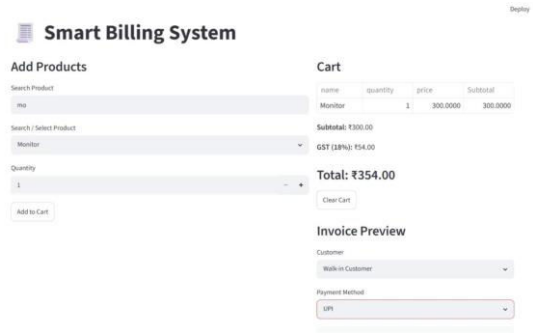


Fig. 3. Billing module: cart, GST calculation, and invoice preview.

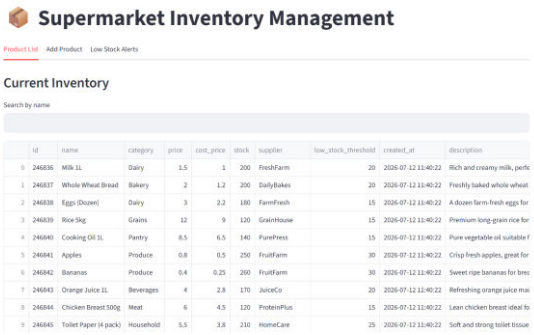


Fig. 4. Inventory module: current stock listing with supplier and threshold data.

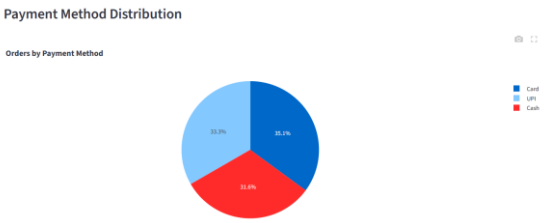


Fig. 5. Analytics module: payment-method distribution chart.

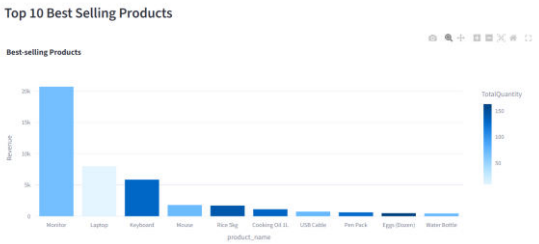


Fig. 6. Analytics module: top-10 best-selling products by revenue.

VII. DISCUSSION AND LIMITATIONS

While Smart POS successfully met its objectives for single-location SMB deployment, several limitations were identified. The recommendation engine relies on predefined rules and market-basket analysis rather than a learning model, so it does

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

not automatically adapt to shifting purchasing behaviour and must be reviewed manually as the catalogue evolves. As product catalogues grow or customer preferences shift, recommendation quality depends on how promptly the underlying rules are maintained, which may not scale well for retailers with large or fast-changing inventories.

SQLite, while ideal for a single-terminal deployment, has limited support for high-concurrency, multi-terminal write operations; a client-server database such as PostgreSQL or MySQL would be required for multi-branch deployments involving simultaneous billing terminals. The prototype also lacks role-based access control and user authentication, meaning any user with access to the application can perform all operations — billing, inventory edits, reconciliation, and analytics viewing alike — which may raise accountability concerns in multi-employee settings. Finally, the system operates entirely offline with no built-in cloud synchronization, which is advantageous for uninterrupted operation and data privacy but precludes centralized multi-branch management and remote monitoring without further development. None of these limitations, however, significantly affect the system's suitability for its intended use case of a single, small retail location.

A further limitation is functional scope: the current prototype does not implement several features common in commercial retail systems, including barcode-scanner integration, QR-code or digital-wallet payments, supplier and purchase-order management, employee attendance tracking, automated invoice emailing, or predictive demand forecasting. These were intentionally left out of scope to keep the project achievable within an internship timeframe, but the modular architecture was chosen so such features could be added later with minimal disruption.

VIII. CONCLUSION AND FUTURE WORK

This paper presented Smart POS, a self-hosted, zero-transaction-fee point-of-sale system that integrates billing, inventory management, rule-based product recommendation, sales analytics, and financial reconciliation into a single Streamlit-based platform built entirely on open-source technologies. Testing across unit, integration, functional, and acceptance levels confirmed that the system delivers accurate billing, reliable inventory tracking, relevant recommendations, and real-time analytics under typical retail workloads. The results support the broader conclusion that appropriate technology selection — SQLite instead of a heavier DBMS, Streamlit instead of a full-stack web framework, and deterministic rule-based recommendation instead of machine learning — can yield a system that is both functionally competitive with commercial alternatives and economically accessible to small businesses.

Future work will focus on replacing the rule-based recommendation engine with a collaborative-filtering or association-rule-learning model that adapts continuously to purchasing behaviour, migrating the backend to a client-server database such as PostgreSQL to support multi-terminal and multi-branch concurrency, and implementing role-based access control with secure authentication and audit logging. Additional planned enhancements include barcode and QR-code integration, digital payment support, customer loyalty and CRM features, AI-assisted demand forecasting, automated cloud backup, and companion mobile applications for remote monitoring — all of which can be incorporated within the existing modular architecture with minimal structural change.

More broadly, this work illustrates that meaningful digital transformation for small retail businesses does not require expensive proprietary platforms or complex cloud infrastructure. By favouring lightweight, well-understood open-source components at every design decision, the Smart POS project shows that a single developer can produce a system automating the core operational needs of a small shop — billing, inventory, recommendations, analytics, and reconciliation — while keeping development complexity and total cost of ownership low.

IX. ACKNOWLEDGMENT

The author gratefully acknowledges the guidance of Dr. B Rajesh Kumar, Department of Software Systems, Sri Krishna Arts and Science College, Coimbatore, throughout the design, development, and evaluation of the Smart POS system, as well as the support of the Department of Software Systems in providing the academic framework for this internship project.

ISSN: 2582-7219 | www.ijmrset.com | Impact Factor: 9.864| ESTD Year: 2018|

International Journal of Multidisciplinary Research in Science, Engineering and Technology (IJMRSET)

(A Monthly, Peer Reviewed, Refereed, Scholarly Indexed, Open Access Journal)

REFERENCES

[1] Python Software Foundation, "Python Language Reference," Python documentation, 2024.
[2] Streamlit Inc., "Streamlit Documentation," 2024.
[3] SQLite Consortium, "SQLite Documentation," 2024.
[4] The pandas development team, "pandas Documentation," 2024.
[5] Plotly Technologies Inc., "Plotly Python Graphing Library," 2024.
[6] R. Elmasri and S. B. Navathe, Fundamentals of Database Systems, Pearson Education, 2016.
[7] I. Sommerville, Software Engineering, 10th ed., Pearson Education, 2015.
[8] D. M. Kroenke and D. J. Auer, Database Processing: Fundamentals, Design, and Implementation, Pearson Education, 2016.
[9] W. McKinney, Python for Data Analysis, 2nd ed., O'Reilly Media, 2017.
[10] R. S. Pressman, Software Engineering: A Practitioner's Approach, 8th ed., McGraw-Hill Education, 2014.

INTERNATIONAL STANDARD SERIAL NUMBER INDIA

INTERNATIONAL JOURNAL OF MULTIDISCIPLINARY RESEARCH

IN SCIENCE, ENGINEERING AND TECHNOLOGY

| Mobile No: +91-6381907438 | Whatsapp: +91-6381907438 | ijmrset@gmail.com |

www.ijmrset.com