International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 13 Issue: 04 | Apr 2026 www.irjet.net p-ISSN: 2395-0072

AutoLedger: Privacy-Preserving Offline Expense Tracking Using Notification Intelligence

Om Shivade[1], Ashlesh Satpute[2], Ishita Jagkar [3], Kiran Balde [4], Prof. Sujata Tirpude[5]

[1],[2],[3],[4] *Dept. of Computer Engineering, Bharat College of Engineering, Badlapur, Maharashtra, India*
[5] *Assistant Professor, Dept. of Computer Engineering, Bharat College of Engineering Badlapur, Maharashtra, India*

Abstract - *Digital payments in India have seen a sharp rise over the past few years, with UPI transactions alone crossing billions of monthly transfers [1]. Yet despite this growth, most users still rely on manual methods to track where their money goes — spreadsheets, notes apps, or nothing at all [2]. The core problem is not motivation but friction: transaction data is scattered across banking apps, wallets, and SMS inboxes, and no single tool pulls it together without requiring cloud access or invasive permissions [3], [4]. AutoLedger is an Android application built to solve this at the device level. It listens to payment notifications using Android's Notification Listener Service [5], pulls out transaction details through rule-based parsing, and logs everything into an encrypted local database [7], [8] — no internet required, no data leaving the phone. The approach is different from most existing trackers, which either need SMS read access (increasingly restricted on modern Android) or push financial data to remote servers.*

Key Words: **Expense Tracking, Offline Systems, Notification Parsing, Android Application, Data Privacy, Financial Analytics, Rule-Based Categorization.**

1. INTRODUCTION

Paying for something in India today rarely involves cash. UPI has made it possible to split a restaurant bill, pay an auto-rickshaw driver, or transfer rent in under ten seconds [1]. The average urban smartphone user now makes dozens of digital transactions every week across multiple apps — GPay, PhonePe, Paytm, bank apps — each generating its own notification and record.

The problem is that none of these apps talk to each other. A user who wants to know how much they spent on food last month has to manually check four or five different apps, add up the numbers, and hope they did not miss anything [2]. Most people do not bother. This is not a discipline problem — it is a design problem. The tools for spending money are seamless; the tools for understanding spending are not.

Existing attempts at fixing this fall into two camps, both with real drawbacks. SMS-based trackers parse bank messages to log transactions automatically, but Android has progressively tightened SMS permissions since version 6.0, making these apps less reliable with each OS update [3]. Cloud-based finance apps offer clean dashboards and cross-device sync, but they require users to hand over sensitive financial data to third-party servers — a trade-off many users are understandably uncomfortable with [4].

AutoLedger takes a different route. By reading payment notifications through Android's Notification Listener Service [5] rather than SMS, it stays within current permission boundaries. All processing and storage happens on the device itself, using an encrypted local database [7], [8]. The result is an expense tracker that works automatically, works offline, and does not require the user to trust anyone with their data.

2. RELATED WORK

Research in personal finance management systems has explored multiple approaches to automate expense tracking and improve financial awareness.

2.1 SMS-Based Transaction Detection Systems

SMS-based expense tracking was one of the first serious attempts at automating personal finance logging on Android. The core idea was straightforward — bank-generated text messages already contain transaction details, so parsing them removes the need for manual entry entirely [3]. Systems built on this approach used rule-based parsers and, later, classifiers like Naive Bayes to extract amounts, merchant names, and timestamps directly from the message text [2].

In practice, though, this approach started running into a wall around Android 6.0, when Google began tightening READ_SMS permissions. By Android 10, many SMS-reading methods were outright blocked for third-party apps not set as the default messaging application. Banks also keep changing their message formats — a parser tuned to one bank's alerts breaks silently when that bank updates its template. The result is a category of apps that requires frequent maintenance just to stay functional, and even then cannot detect transactions from UPI apps that never send SMS at all.

2.2 Cloud-Based Personal Finance Applications

Cloud-connected finance apps solve the data-fragmentation problem differently — by pulling transactions from multiple sources into a single account and syncing them across devices [1]. Some integrate directly with banking APIs; others use credential-based screen scraping. The user

experience is typically polished, with budgeting dashboards, category breakdowns, and spending trend charts.

The trade-off is that the user has to hand their financial data to a third-party server to get any of that [4]. For users in India where data privacy regulation around fintech is still evolving, that is a meaningful concern. There is also a practical reliability issue — these apps stop working without an internet connection, which matters in areas with inconsistent mobile data coverage. A system that fails precisely when a user is in a market or travelling is not much use as a daily tracker.

2.3 Notification-Based Transaction Detection

Notification-based detection emerged as a middle path — it captures transaction alerts from banking apps in real time without needing SMS access or banking API credentials [6]. Android's NotificationListenerService was designed for accessibility tools, but it works equally well for intercepting payment confirmations from apps like GPay, PhonePe, or HDFC MobileBanking.

Research in this area, including Shaik et al. [3], has shown that notification parsing can match SMS-based detection coverage while requiring far fewer sensitive permissions. The limitation identified in most implementations is that the detection layer is paired with a cloud backend for storage and analysis — which partially defeats the privacy argument for avoiding SMS access in the first place. AutoLedger specifically addresses this by keeping the entire pipeline, from notification capture to storage, on-device.

2.4 Automated Expense Categorization Systems

Categorization is where most research effort has gone. Thakare et al. [2] demonstrated that Naive Bayes classifiers can classify expense transactions from notification text with reasonable accuracy when trained on labeled data. Other studies have applied Random Forest and SVM-based approaches to the same problem, reporting accuracy improvements at the cost of larger model sizes and less interpretable outputs [3].

The practical issue with ML-based categorization for an offline, privacy-first app is the bootstrapping problem — you need training data to build the model, and collecting that data requires either a cloud pipeline or users to label their own transactions. Rule-based categorization sidesteps this entirely. It is less adaptive, but for a well-defined set of Indian merchants and spending categories, a curated keyword-to-category mapping works reliably without any data collection, model training, or inference overhead [4].

2.5 Gap and Positioning

What is missing across these categories is a system that combines automatic detection, full offline operation, and transparent categorization in a single app without routing financial data through external servers. SMS-based systems fail on permission grounds; cloud-based systems fail on privacy grounds; notification-based systems mostly fail by still depending on cloud backends; and ML-based systems introduce complexity that does not fit a lightweight offline tool.

AutoLedger is designed to close all four gaps simultaneously — notification-based detection through Android's NotificationListenerService [6], rule-based categorization that runs entirely on-device [2], encrypted local storage via Room with SQLCipher [7], [8], and supplementary manual entry and receipt scanning for transactions that generate no digital alert at all. A comparative analysis of existing expense tracking approaches is presented in Table -1

Table -1: Comparative Analysis of Expense Tracking Approaches

Approach	Key Feature	Limitation
Manual Tracking [2]	Full user control	Time-consuming
SMS-Based [3]	Automatic logging	Permission restrictions
Cloud-Based [4]	Sync & analytics	Privacy & internet dependency
ML-Based [9]	Accurate categorization	Complex, less transparent
Notification-Based [6]	Real-time detection	Often cloud-dependent
AutoLedger (Proposed)	Offline, secure, automated	Limited adaptability

3. SYSTEM ARCHITECTURE

The AutoLedger system is built around a single design constraint: nothing leaves the device. Every module — from notification capture to data storage to report generation — runs locally on the user's Android phone. This is a deliberate departure from how most expense tracking apps are built, where the device is essentially a thin client that ships data to a cloud backend for processing [4].

The architecture is modular, with each component handling one well-defined job and passing structured data to the next. Fig. 1 shows the full pipeline. Three input sources feed into the system — the Notification Listener Service for automatic digital transaction detection, the Manual Expense Entry module for cash and offline payments, and the Receipt & Bill Scanning module for physical receipts. All three converge at the Transaction Parsing layer, flow through the Rule-Based Categorization Engine, and land in the Encrypted Local Database. The User Interface layer sits on top of the database and handles everything the user actually sees.

3.1 Notification Listener Service

This is where the system starts. Android's NotificationListenerService [5] is a system-level API that lets a permitted application read notifications as they arrive on the device — the same mechanism used by accessibility tools and smartwatch companion apps. AutoLedger registers itself as a notification listener and watches specifically for alerts from banking and UPI applications.

When a payment goes through on GPay, PhonePe, or a bank app, a confirmation notification fires within seconds. The listener intercepts that notification text and forwards it to the parsing module. No SMS permission is needed, no banking API credentials are required, and crucially, the raw notification text never leaves the device. This approach works across any app that generates payment confirmations — which in practice covers the entire Indian UPI ecosystem [3].

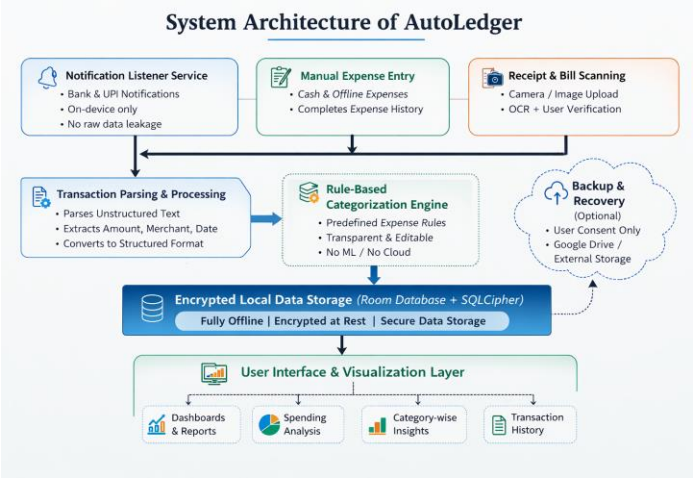

Fig -1: The overall system architecture

3.2 Transaction Parsing and Processing Module

Raw notification text from different banks looks very different. An HDFC alert reads differently from an SBI one, and a GPay confirmation has a different structure from a Paytm one. The parsing module handles this variability using a rule-based text analysis mechanism that scans for known keywords and numerical patterns.

For each incoming notification, the module attempts to extract five key attributes: transaction amount, merchant or receiver name, date and time, payment method (UPI, card, or bank transfer), and transaction status. Keywords like "debited", "paid", "credited", and "transaction successful" anchor the extraction logic. Once identified, these attributes are assembled into a standardized transaction object — the same structure regardless of which bank or app generated the original alert. This standardization is what allows the rest of the pipeline to work consistently [6].

3.3 Rule-Based Categorization Engine

Once a structured transaction object arrives at the categorization engine, it gets assigned to one of the predefined expense categories: food, transportation, shopping, utilities, entertainment, and so on. The engine does this by matching the merchant name or transaction description against a stored keyword table [9].

A transaction from Swiggy or Zomato maps to food. A payment at a fuel station maps to transportation. A transfer to an online shopping platform maps to shopping. The mapping is explicit, deterministic, and inspectable — if a transaction lands in the wrong category, the reason is traceable and fixable. This is a conscious choice over ML-based categorization, which can produce correct results but through logic that is difficult for a user to understand or correct [2]. The rule base can be extended in future versions as new merchants and payment patterns emerge.

3.4 Manual Expense Entry Module

Not every expense generates a notification. Cash paid to a vegetable vendor, a rickshaw fare, or a chai at a local stall — none of these produce digital alerts. The manual entry module exists specifically for these cases. Users can log an expense by entering the amount, selecting a category, picking a date, and optionally adding a note. These records are stored in exactly the same database structure as automatically detected transactions, so the reporting layer treats them identically [2].

This is what makes AutoLedger a complete expense tracker rather than just a UPI monitor. Without manual entry, the financial picture would be systematically incomplete for anyone who still uses cash for everyday small purchases — which describes most users in semi-urban India.

3.5 Receipt and Bill Scanning Module

The receipt scanning module handles a third category of expense — physical bills from shops, restaurants, or service providers where a digital payment was made but the notification was missed or not generated. Users can photograph a receipt with the device camera or upload an image from their gallery.

The module extracts the relevant details — primarily the total amount and merchant name — and presents them to the user for confirmation before saving. The confirmation step is intentional: OCR accuracy depends on image quality and receipt layout, and a wrong amount saved silently is worse than no entry at all [3]. After the user confirms, the record enters the database through the same path as any other transaction.

3.6 Encrypted Local Data Storage

All transaction records — whether auto-detected, manually entered, or scanned — are stored in a Room Database [7] encrypted with SQLCipher [8]. Room provides a structured, query-friendly abstraction over SQLite that fits naturally into Android's architecture components. SQLCipher adds full database encryption, meaning the database file itself is unreadable without the correct key even if someone extracts it directly from device storage.

The schema organizes records into tables with fields for transaction ID, merchant name, category, amount, timestamp, payment method, and transaction type (debit or credit). All reads and writes happen locally with no network calls involved, which also means retrieval is fast — there is no round-trip latency to a remote server.

3.7 User Interface and Visualization Layer

The UI layer is built on Android XML with Material Components and organized around a single-activity, multi-fragment navigation pattern. HomeFragment displays the current transaction list and total spent. ReportsFragment generates daily, weekly, and monthly spending breakdowns by category. ScanFragment handles the camera and image upload flow for receipt scanning. SettingsFragment manages permissions, app lock configuration, and backup preferences.

The dashboard gives users a category-wise spending breakdown that updates in real time as new transactions are recorded. The goal is not just to log expenses but to make spending patterns visible — because awareness is what drives better financial decisions [1].

3.8 Backup and Recovery Module

Since all data lives on the device, losing the device means losing the data unless a backup exists. The backup module lets users export an encrypted copy of the database to Google Drive or share it via email. The key word is "lets" — backup is never automatic without the user initiating it. This keeps the system's offline-first, consent-first design consistent: no data moves anywhere the user has not explicitly directed it to go [4].

Restoring from backup reverses the process — the encrypted file is imported and decrypted into the local database, recovering the full transaction history.

3.9 Use Case Diagram

Fig. 2 shows the use case diagram for AutoLedger. There are two actors in the system — the User and the Android System. The Android System is a supporting actor that the user never directly interacts with; it exists in the diagram because three things the app depends on — notification delivery, biometric authentication, and Google Drive backup — are services the OS provides rather than things AutoLedger implements itself [5].

The User's interactions branch into eight primary use cases. Auto Detect Expense is triggered by the Android System forwarding a bank or UPI notification, after which the user can View Expenses from the logged record. Add Expense Manually covers cash and offline transactions the user enters directly. Scan Bill and Upload Screenshot both feed the receipt scanning pipeline for transactions not captured through notifications. View Reports extends into three sub-cases — Daily Report, Weekly Report, and Monthly Report — giving users spending breakdowns at different time granularities.

The two remaining use cases cover security and data management. Enable App Lock extends into Unlock via PIN and Unlock via Biometric, both of which depend on the Android System's authentication services. Backup Data and Restore Backup are paired operations — backup writes an encrypted copy of the database to Google Drive, restore reads it back. Both are user-initiated; neither runs automatically [7], [8].

What the diagram makes clear is that AutoLedger is designed around two distinct usage modes that coexist in the same app: a fully automatic mode where the system runs in the background and logs transactions without any user action, and a manual mode where the user can audit, supplement, and manage their financial records directly.

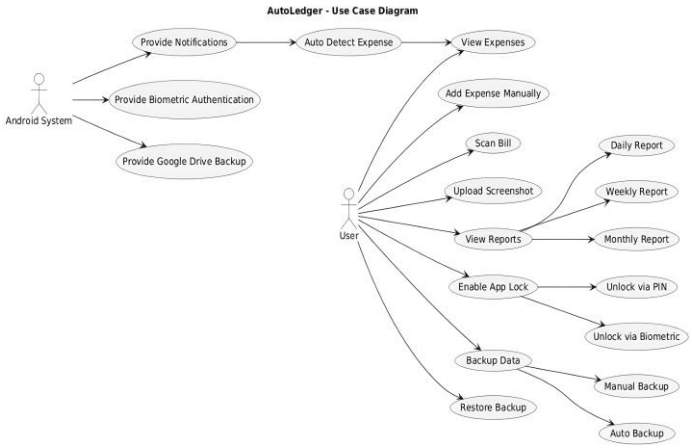

Fig -2: Use Case Diagram of AutoLedger System

3.10 Sequence Diagram

Fig. 3 traces the component interaction for the receipt scanning flow, which is the most involved of the three input paths because it crosses the most module boundaries. The sequence starts when the User opens the app through Main Activity and navigates to the scanning interface. From there, the user captures a bill image or uploads one from the gallery — both paths reach ScanFragment, which hands the image off for processing.

ScanFragment passes the input to TransactionParser. This is the module doing the actual work of reading the image

content and pulling out the financial details — amount, merchant, and timestamp. Once TransactionParser has produced a structured transaction object, it sends that to CategoryEngine, which runs the keyword-matching logic and returns an assigned expense category. The round trip between TransactionParser and CategoryEngine produces a complete, categorized transaction record [9].

That record then goes to AutoLedgerDatabase for storage. Room handles the write operation with SQLCipher encryption applied transparently at the storage layer [7], [8]. AutoLedgerDatabase returns a success confirmation, and on that signal, ReportsFragment triggers a data refresh — pulling the updated transaction list and recalculating the spending summaries displayed on the dashboard.

The entire sequence — from image capture to the dashboard updating — happens on-device with no external calls. The user sees their new transaction appear in the reports within seconds of confirming it [6].

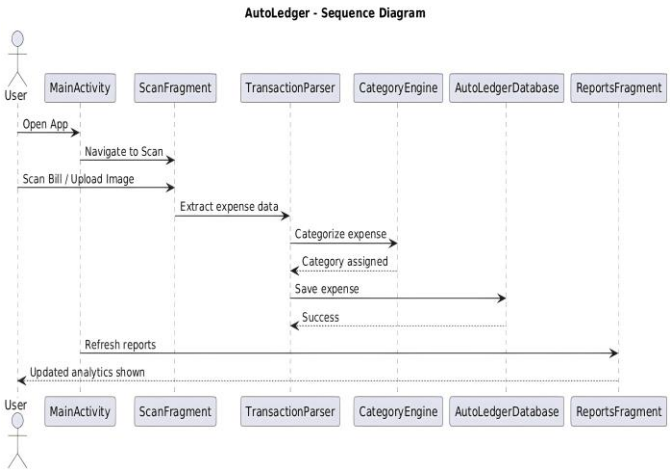

Fig -3: Sequence Diagram of AutoLedger System

3.11 Class Diagram

Fig. 4 shows the class structure of the AutoLedger application. The design follows a single-activity, multi-fragment pattern with MainActivity at the top of the navigation hierarchy. MainActivity owns two methods — onCreate() and loadFragment() — and is responsible for routing the user between the four main fragments: HomeFragment, ScanFragment, ReportsFragment, and SettingsFragment.

HomeFragment exposes showExpenses() and showTotalSpent(), giving the user their current transaction list and a running total. ReportsFragment extends this with showDailyReport(), showWeeklyReport(), and showMonthlyReport() — three time-granularity views built from the same underlying data. ScanFragment handles the camera and upload interface through openCamera() and uploadImage().

The processing side of the diagram sits separately from the UI side. TransactionParser receives notification or image data and runs three extraction methods — parseNotification(), extractAmount(), and extractMerchant() — to produce a structured transaction object. That object flows to CategoryEngine, which has a single public method categorizeExpense() that applies the keyword ruleset and returns a category assignment [9].

Storage is handled by AutoLedgerDatabase and ExpenseDao working together. AutoLedgerDatabase is a singleton accessed through getInstance() and exposes the DAO through expenseDao(). ExpenseDao implements the three core database operations: insertExpense(), getAllExpenses(), and deleteExpense(). The entity stored by these operations is ExpenseEntity, which carries six fields — id, title, amount, category, timestamp, and isDebit [7], [8].

The security and backup classes complete the diagram. LockActivity manages the app lock screen and exposes verifyPIN() and authenticateBiometric(). SecurityPrefs stores the lock configuration state through isAppLockEnabled(), savePIN(), and verifyPIN(). BackupManager handles the three backup operations: manualBackup(), autoBackup(), and restoreBackup() — though autoBackup() only runs when the user has explicitly scheduled it, not silently in the background [4].

The class structure reflects a clean separation of concerns: UI fragments do not touch the database directly, processing classes do not know about the UI, and security is isolated in its own layer. This separation makes each component independently testable and straightforward to extend in future versions [1].

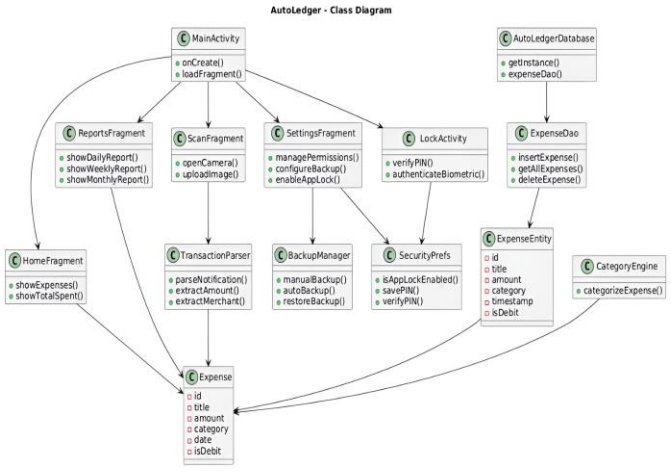

Fig -4: Class Diagram of AutoLedger System

4. METHODOLOGY

The AutoLedger system follows a structured workflow to automatically capture, process, and store financial transactions generated through banking and UPI applications. Process begins with real-time detection of

transaction notifications using Android's Notification Listener Service [5]. These notifications are analyzed to extract essential financial details such as transaction amount, merchant name, date, and payment method using rule-based parsing techniques.

The extracted information is then transformed into a standardized format and classified into predefined expense categories based on keyword and merchant-based rules [9]. This approach ensures consistent and transparent categorization without relying on complex machine learning models.

All processed transaction data is securely stored in a local database implemented using Room with SQLCipher encryption, ensuring data privacy and offline accessibility [7], [8]. In addition to automated tracking, the system supports manual expense entry and receipt scanning to capture transactions that are not detected through notifications.

Receipt scanning covers the gap between the two — digital payments where the notification was missed, or card transactions that produced a paper receipt. The user photographs or uploads a receipt image through ScanFragment; TransactionParser applies OCR extraction to pull the amount and merchant; the result is shown in a confirmation screen before anything is saved [3]. The confirmation step exists because OCR accuracy depends heavily on image quality — a wrong amount confirmed by the user is acceptable, a wrong amount stored silently is not.

Finally, the stored data is presented through dashboards and reports, enabling users to analyze spending patterns and gain meaningful financial insights. The overall workflow of the proposed system is illustrated in Fig -5.

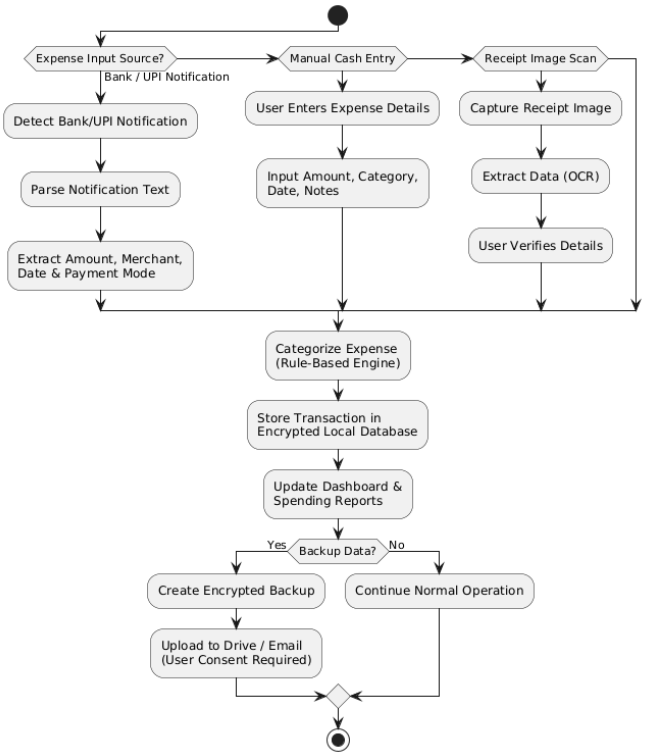

Fig -5: Workflow of the Proposed System

5. EXPERIMENTAL SETUP

The AutoLedger system was evaluated under practical usage conditions to assess its functionality, reliability, and performance. Since the system operates without predefined datasets, testing was conducted using real-time financial transactions generated through various banking and UPI applications. Multiple transactions were performed to ensure that the system could accurately capture notifications and handle variations in message formats [6].

The application was developed using Java in Android Studio, with the user interface designed using XML and Material Components. Transaction detection was implemented using Android's Notification Listener Service [5], while data storage was handled locally using Room Database with SQLCipher encryption to ensure data security [7], [8]. The system was tested on Android devices running version 9.0 and above, with a minimum of 4 GB RAM and standard hardware configurations.

Evaluation focused on key functional aspects, including accuracy of notification detection, correctness of transaction data extraction, reliability of expense categorization, and overall system performance. The rule-based categorization mechanism ensured consistent classification without the need for training data, maintaining transparency and efficiency [9]. Additional testing was performed for manual expense entry and receipt scanning to verify that the system supports complete expense tracking.

The results indicate that the system performs efficiently with minimal processing delay, as all operations are executed locally without reliance on cloud services. This approach reduces latency and enhances system responsiveness compared to cloud-based expense tracking solutions [4]. The application maintained stable performance across different scenarios, demonstrating its suitability for real-world personal finance management.

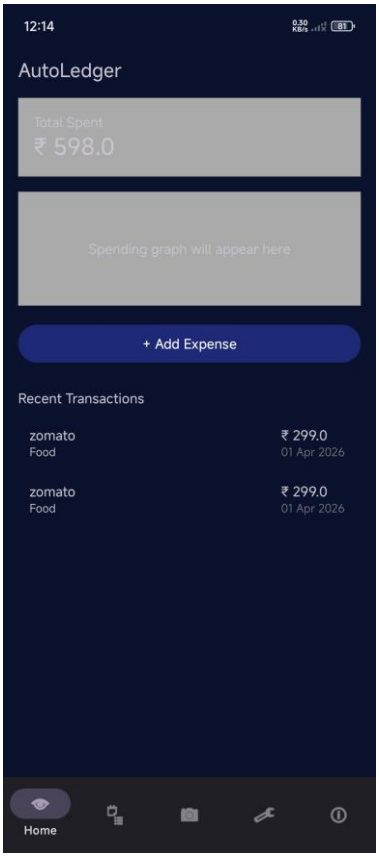

Fig -6: Main dashboard of AutoLedger showing total expenditure

Fig -7: Reports module displaying category-wise expense breakdown

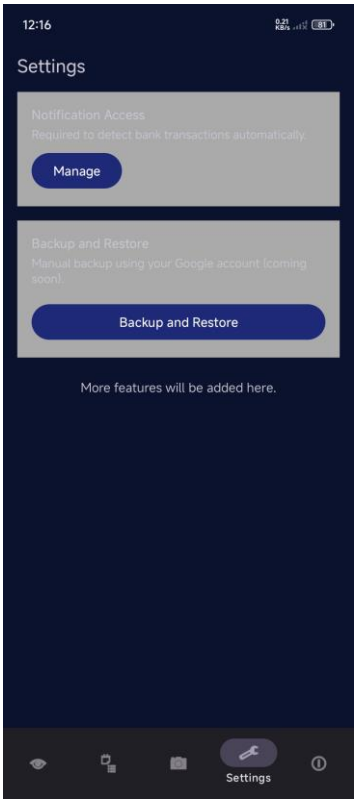

Fig -8: Settings interface showing notification access management

6. RESULTS AND DISCUSSION

The AutoLedger system demonstrated reliable performance during real-world testing, successfully capturing and processing financial transactions from multiple banking and UPI applications. The notification-based detection mechanism consistently identified transaction events and extracted relevant details such as amount, merchant name, and timestamp with high accuracy.

The rule-based categorization approach provided consistent and transparent classification of expenses into predefined categories. While machine learning-based systems offer adaptive learning capabilities, the proposed approach avoids complexity and ensures predictable results, making it suitable for offline environments [9].

In terms of performance, the system operates efficiently as all processing is performed locally on the device. This eliminates delays associated with network communication and ensures smooth execution even on mid-range smartphones. The use of Room Database with SQLCipher encryption ensures secure storage of financial data without compromising access speed [7], [8].

The integration of additional features such as manual expense entry and receipt scanning ensures comprehensive expense tracking, covering both digital and offline transactions. Furthermore, the reporting module provides meaningful insights through categorized summaries, enabling users to better understand their spending patterns.

Overall, the results indicate that AutoLedger achieves a balance between automation, privacy, and performance. Compared to cloud-based expense tracking systems, the proposed solution offers enhanced data security and operational independence while maintaining usability and efficiency [4].

7. CONCLUSION

This paper presented AutoLedger, an offline-first expense tracking system that automates the process of recording and managing financial transactions using notification-based detection. By leveraging Android's Notification Listener Service, the system eliminates the need for manual data entry and reduces dependency on SMS parsing or cloud-based solutions.

The proposed approach ensures that all transaction data is processed and stored locally using secure database mechanisms, enhancing user privacy and system reliability. The integration of rule-based categorization enables transparent and consistent classification of expenses without introducing computational complexity.

Experimental results demonstrate that the system performs efficiently across different usage scenarios, providing accurate transaction detection and meaningful financial insights. Additional features such as manual entry and receipt scanning further improve the completeness of expense tracking.

The system offers a practical and secure solution for personal financial management by combining automation, privacy, and usability within a unified framework.

8. FUTURE WORK

Future enhancements to the AutoLedger system can focus on improving adaptability, intelligence, and user experience. One possible extension is the integration of lightweight on-device machine learning models to enable adaptive expense categorization based on user behavior, while still maintaining privacy.

The system can also be enhanced by incorporating advanced receipt recognition using Optical Character Recognition (OCR) techniques to improve accuracy in extracting data from physical bills. Additionally, expanding support for multiple languages and diverse notification formats can improve usability across a wider range of users and financial platforms.

Another area of improvement is the implementation of secure cross-device synchronization using end-to-end encryption, allowing users to access their financial data across multiple devices without compromising privacy. Further enhancements may include predictive analytics and personalized financial insights to assist users in budgeting and spending decisions.

These improvements can extend the capabilities of AutoLedger, making it more intelligent, scalable, and adaptable to evolving user needs.

ACKNOWLEDGMENT

The authors sincerely thank Prof. Sujata Tirpude and the Department of Computer Engineering, Bharat College of Engineering, for their guidance and support.

AI DISCLOSURE

The authors acknowledge the use of AI-assisted tools for language refinement, formatting, and structuring of the manuscript. All technical content, system design, and implementation details are based on the authors' original work and understanding.

REFERENCES

[1] T. Stefanov, M. Stefanova, S. Varbanova and S. Temelkov, "Personal Finance Management Application," TEM Journal, vol. 13, no. 3, pp. 2066-2075, Aug. 2024, doi: 10.18421/TEM133-34.

[2] S. A. Sabab, S. S. Islam, M. J. Rana and M. Hossain, "eExpense: A Smart Approach to Track Everyday Expense," 2018 4th International Conference on Electrical Engineering and Information & Communication Technology (iCEEiCT), Dhaka, Bangladesh, 2018, pp. 136-141, doi: 10.1109/CEEICT.2018.8628070.

[3] S. Shaik, A. S. R. Mikkili, M. R. Nandyala and D. R. Namani, "Automating Financial Management: An Exploration of Automatic Expense Tracking Systems," in Intelligent Systems with Applications in Communications, Computing and IoT, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 621, Springer, Cham, 2025, pp. 270-282, doi: 10.1007/978-3-031-92614-3_20.

[4] X. T. Koo and K. C. Khor, "Expense Tracking with Tesseract Optical Character Recognition v5: A Mobile Application Development," 2023 IEEE Symposium on Industrial Electronics & Applications (ISIEA), Kuala Lumpur, Malaysia, 2023, pp. 1-5, doi: 10.1109/ISIEA58478.2023.10212265.

[5] Android Developers, "NotificationListenerService," Available:https://developer.android.com/reference/android/service/notification/NotificationListenerService

[6] S. Shaik, A. S. R. Mikkili, M. R. Nandyala and D. R. Namani, "Automating Financial Management: An Exploration of Automatic Expense Tracking Systems," in Intelligent Systems with Applications in Communications, Computing and IoT, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, vol. 621, Springer, Cham, 2025, pp. 270-282, doi: 10.1007/978-3-031-92614-3_20.

[7] Android Developers, "Room Persistence Library," Available: https://developer.android.com/training/data-storage/room

[8] Zetetic LLC, "SQLCipher: SQLite Database Encryption," Available: https://www.zetetic.net/sqlcipher/

[9] R. Thakare, N. Thakare, R. Sangtani, S. Bondre and A. Manekar, "Expense Tracker Application using Naive Bayes," SSRG International Journal of Recent Engineering Science, vol. 10, no. 3, pp. 50-56, 2023, doi: 10.14445/23497157/IJRES-V10I3P108.