1. Introduction

Problem Statement:
Tracking expenses across multiple sources, such as SMS notifications, is a common challenge. Many users receive numerous SMS notifications related to financial transactions (e.g., bank withdrawals, card payments), making it difficult to compile and categorize these transactions for budgeting or financial tracking purposes.

Solution Overview:

Initial Approach: Regex-Based Extraction

At first, I used **regular expressions (regex)** to extract expense-related information from SMS messages. Regex provides a way to identify and parse specific - patterns such as amounts, dates, and keywords like "debited" or "credited" from messages.

However, regex had some **limitations** for this use case:

- **Rigidity**: Regex requires precise pattern definitions. SMS messages can vary significantly in format, making it hard to account for all variations.
- **Limited Contextual Understanding**: Regex lacks the ability to understand context, which is crucial for interpreting natural language and extracting nuanced information.

Code Snippet:

```
const PATTERNS = {

amount: /(?:INR|Rs\.?)\s*([0-9,]+\.?\d*)/i,

merchant:
/(?:at|to|in)\s+([A-Za-z0-9\s&'+\-.,()]+?)(?:\s+on|info|Ltd|Pvt|limited|vs|\.|$)/i,

debit: /(?:debited|spent|paid|withdrew|withdrawal|purchase)/i,

credit: /(?:credited|received|refund|cashback)/i

};
```

Transition to an LLM for Enhanced Accuracy

To overcome these limitations, I shifted to an **open-source language model (LLM) Gemini** for data extraction. The LLM approach offered several improvements:

- **Better Contextual Understanding (NER)**: Unlike regex, the LLM can understand contextual clues within messages, such as distinguishing between multiple amounts or recognizing implicit information about the expense.
- **Scalability**: The LLM approach is easier to maintain and scale since it doesn't rely on predefined patterns. Adding new categories or information types requires minimal changes.

Code Snippet:

```
const { GoogleGenerativeAI } = require("@google/generative-ai");

const genAI = new GoogleGenerativeAI(API_KEY);

const model = genAI.getGenerativeModel({ model: "gemini-1.5-flash" });


 const prompt = `Extract the following details from this list of SMS messages. Return each message as a JSON object with this schema:


MessageSchema = {

  "amount": String,  // Format: "+ ₹500.00" for credit, "- ₹500.00" for debit

  "description": String,  // Name of merchant/payer/receiver

  "date": String,  // Format: "YYYY-MM-DD"

  "bank": String   // Bank name involved in the transaction

}

Rules for extraction:

1. Amount:

   - For debited/spent amount, prefix with "-"

   - For credited/received amount, prefix with "+"

   - Always include the ₹ symbol
```

```
   - Maintain 2 decimal places
2. Description:
   - Extract only the merchant/person name
   - Remove UPI IDs, reference numbers, and other technical details
3. Date:
  - Convert all dates to YYYY-MM-DD format
4. Bank:
   - Extract the bank name from transaction details
   - For UPI, use the last bank code (e.g., HDFC, SBI, ICICI)
   - For direct transfers, mention both sender and receiver banks if available

Example Input Messages:
"UPI/DR/430763282641/RAJENDRA KUMAR/HDFC/Rs.7,200.00 debited on 02-11-2024"
"IMPS/CR/RAHUL SHARMA/ICICI/Rs.1,500.00 credited to your account on 02-11-2024"
Return: Array<MessageSchema>
Messages:
${text}`;
const result = await model.generateContent(prompt);
const result1 = result.response.text();
const cleanedResponse = result1.replace(/```json\n?|\n?```/g, '');
const parsedJson = JSON.parse(cleanedResponse.trim());
```

The integration of an LLM allowed us to improve both **accuracy** and **reliability** in data extraction. This enhancement was particularly beneficial for handling the varied formats of transactional SMS messages across different banks and financial institutions.

2. Technical Setup

Standalone React Native Setup:
This project is built using standalone React Native, chosen for its flexibility and compatibility with native device features not fully supported by Expo.

- **Initializing the Project with React Native CLI**
  First, create a new React Native project using the React Native CLI:
  ```
  npx react-native init ExpenseTrackerApp
  ```

  This command initializes a new React Native project named `ExpenseTrackerApp`. The CLI handles creating the project structure, downloading dependencies, and setting up the initial codebase.

- **Configuring Android Studio CLI Tools**
  To enable the Android SDK and tools to work with the project, follow these steps:
  - **Install Android Studio CLI Tools**: Make sure Android Studio is installed. Then, add the Android SDK path to your system's environment variables to allow React Native to locate and use Android tools.

    ```
    brew install node

    brew install watchman

    brew install openjdk@11
    ```

  - **Setting Up Environment Variables:** In your shell configuration file (e.g., `~/.bashrc` or `~/.zshrc`), add:

    ```
    export ANDROID_HOME=$HOME/Library/Android/sdk

    export PATH=$PATH:$ANDROID_HOME/emulator

    export PATH=$PATH:$ANDROID_HOME/tools

    export PATH=$PATH:$ANDROID_HOME/tools/bin
    ```

```
export PATH=$PATH:$ANDROID_HOME/platform-tools
```

Enabling USB Debugging and Verifying Device Connection

To test on a physical Android device, enable **USB Debugging** on the device (found in Developer Options). Then, connect your device via USB.

- **Verify Device Connection**:
  Use the following command to ensure the device is recognized by Android Debug Bridge (ADB):

```
adb devices

->List of devices attached

1234567abcdef device
```

Running the App on the Physical Device:

```
npm run-android
```

It builds and deploys the app on the connected device.

Install Core Dependencies:

```
npm install react react-native
```

Install Project-Specific Libraries:

```
npm install @google/generative-ai \
@react-native-masked-view/masked-view \ @react-navigation/native \
@react-navigation/stack \ date-fns \ react-native-gesture-handler
\ react-native-get-sms-android \ react-native-logs \
react-native-safe-area-context \ react-native-screens \
react-native-vector-icons
```

3. Permissions and Security

SMS Permissions:

- The app requires permission to read SMS messages on the device to identify and parse transaction-related notifications. Upon launch, the app prompts the user to grant SMS reading permissions, using React Native's permissions management library to request and handle permissions in compliance with Android's security policies.
  We need to add this lines in AndroidManifest.xml:
  <uses-permission android:name="android.permission.READ_SMS" />
  <uses-permission android:name="android.permission.RECEIVE_SMS" />

Code Snippet:

```
import { PermissionsAndroid, Platform } from 'react-native';

export const requestSMSPermission = async () => {
 if (Platform.OS === 'android') {
   const granted = await PermissionsAndroid.request(
     PermissionsAndroid.PERMISSIONS.READ_SMS,
       {
       title: "SMS Permission",
       message: "This app needs access to your SMS messages to track expenses.",
       buttonNeutral: "Ask Me Later",
       buttonNegative: "Cancel",
       buttonPositive: "OK"
     }
   );
   return granted === PermissionsAndroid.RESULTS.GRANTED;
 }
 return true;
```

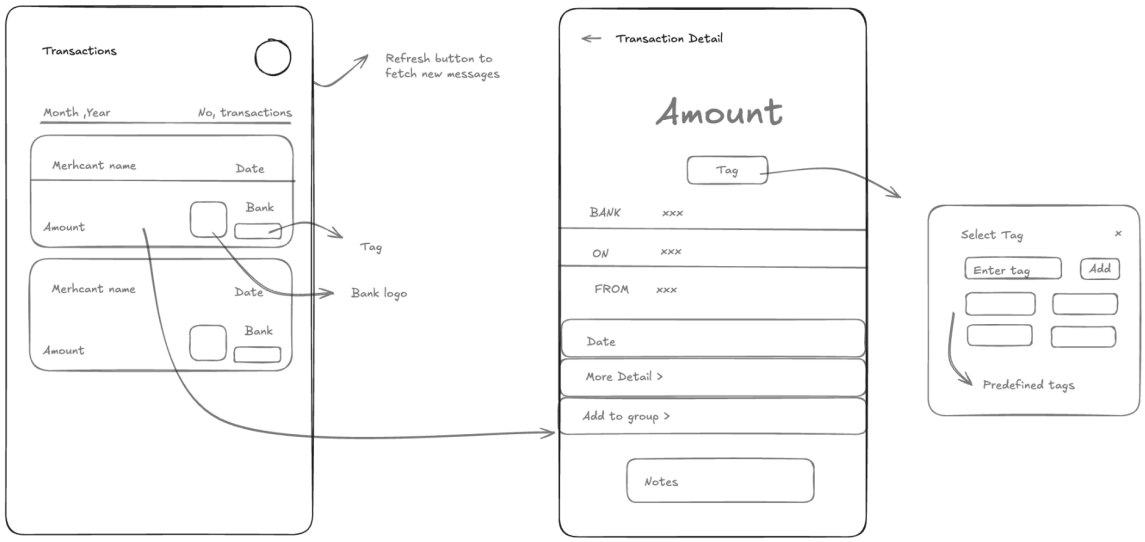
```
};
```

Data Security Considerations:

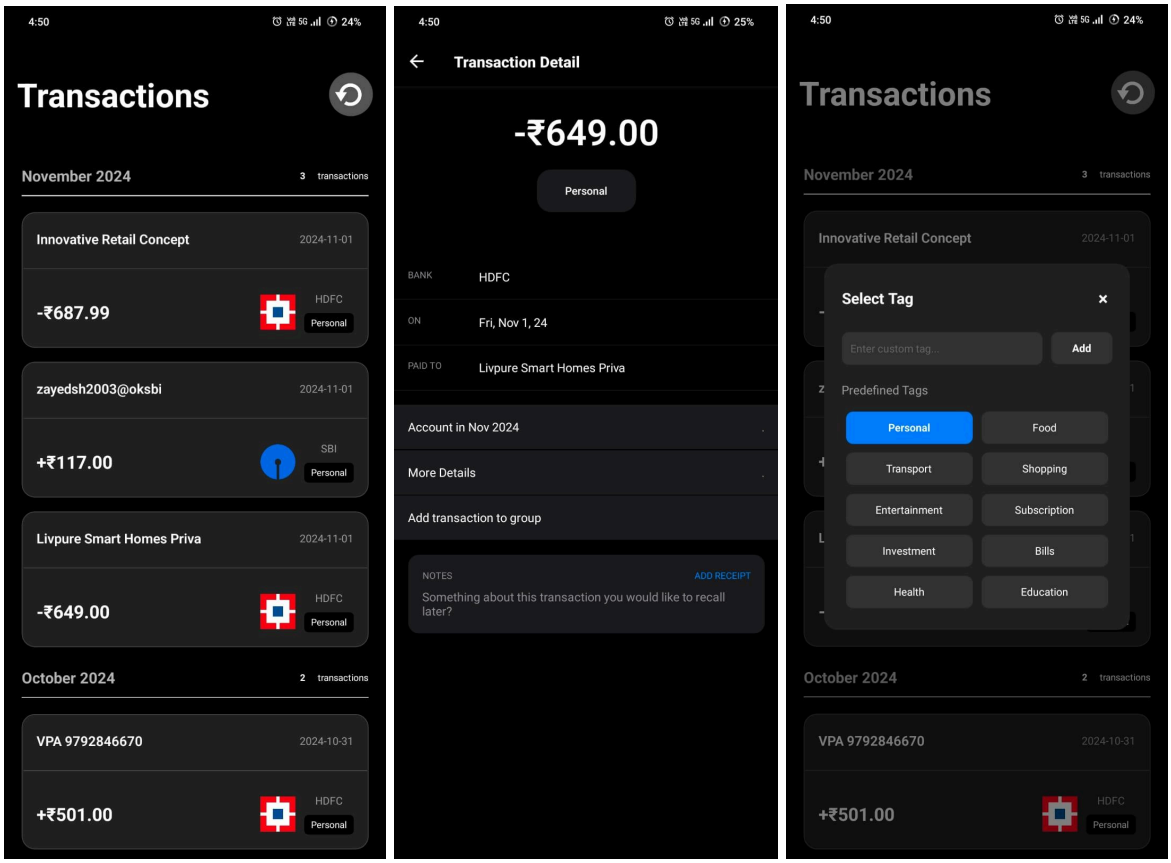
- **Sensitive Data**: SMS messages often contain personal financial information, so user privacy is a priority. While this demonstration used an open-source LLM through an API for data extraction, a local LLM model would ideally replace the remote one to enhance both data security and processing efficiency. However, this was not implemented fully due to device limitations, but it remains a recommended future enhancement.
- **Future Security Enhancements**: Potential future improvements could include fully on-device processing using a locally stored LLM model, encryption of extracted data before storage, and enhanced permissions prompts to reassure users about data privacy.

4. UI Design and User Interaction

Wireframe:



Screenshots:



Testing

Testing Setup:

Testing was conducted on a physical Android device connected via USB debugging, as this setup allowed for real-time interaction and debugging. Using Android Studio's CLI tools and adb, to verify the device connection for ensuring smooth communication between the app and the hardware. Physical device testing was critical because the Expo framework lacks native support for SMS reading, making it unsuitable for this application.

Testing on Additional Devices: Testing was extended to various devices with different screen sizes and Android versions to ensure consistent UI and functionality across platforms. This included testing the SMS reading, transaction extraction, and tagging features.

Challenges Encountered:

Expo Limitations: Due to Expo's restrictions on accessing certain device-level features, we opted for a standalone React Native setup.

Conclusion and Future Enhancements

Summary of Achievements:
This app successfully solves the problem of tracking expenses from SMS notifications by extracting transaction information (e.g., amount, date, description) and displaying it in an organized, easy-to-use interface. The combination of regex filtering and an open-source language model (LLM) enhances information extraction accuracy, making it possible to categorize expenses from unstructured text. The app also provides users with flexibility through tagging, monthly sorting, and detailed transaction views.

Potential Improvements:

1. Local LLM Integration
2. AI-Powered Categorization and Insights
3. Cross-Platform Compatibility

Github Repo Link:

https://github.com/sandeepangh782/SMS_Reader.git