HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

GRADUATION THESIS

Personal Expense Management Software on Android with Google Assistant

PHẠM THÁI DƯƠNG
Duong.PT207595@sis.hust.edu.vn

Program: Information Technology (IT - LTU)

Supervisor: Master of Science Lê Bá Vui _____

Signature

Department: Computer Engineering

School: Information and Communications Technology

HANOI, 06/2025

ACKNOWLEDGMENTS

I would like to express my heartfelt gratitude to all those who have supported me throughout my academic journey. To my loved one, thank you for your unwavering love and encouragement. To my family, your sacrifices and constant support have been my foundation. To my friends, your companionship and motivation have kept me going. To my teachers, your guidance and dedication have inspired me to strive for excellence. Lastly, I thank myself for the hard work and determination that have driven me to achieve my best results. Your support and belief in me have been invaluable, and I am profoundly grateful for everything.

PLEDGE

Student's Full Name: Phạm Thái Dương

Student ID: 20207595

Contact Phone Number: 0357542424

Email: Duong.PT207595@sis.hust.edu.vn

Class: IT-LTU 01

Program: Information Technology – Latrobe University

I, Thái Dương, commit that the Graduation Thesis (GT) is my own research work under the guidance of MSc Lê Bá Vui. The results presented in the GT are truthful and are my own achievements, not copied from any other works. All references in the GT, including images, tables, data, and quotations, are clearly and fully cited in the bibliography. I take full responsibility for any violations of the school's regulations concerning plagiarism.

Hanoi, 18/06/2025

(Signature and full name)

ABSTRACT

In the digital age, the demand for efficient personal finance management on smartphones is growing rapidly. Currently, there are several expense management applications for the Android operating system, but they often lack advanced features and seamless user interaction. Many popular applications focus solely on basic functions like expense tracking and budget creation without optimizing for user experience or leveraging voice-based interaction. These limitations highlight the need for a more comprehensive and user-friendly expense management solution.

The approach I chose is to develop a personal expense management application for Android smartphones, aiming to create a product that not only meets basic financial tracking needs but also enhances user experience through an intuitive interface and voice-activated features powered by Google Assistant. The reason for choosing this approach is the widespread adoption of the Android platform and the potential to integrate advanced technologies like voice recognition for personalized financial management.

My solution involves developing an expense management application with a modern, easy-to-use interface and advanced features such as expense categorization, budget planning, transaction history tracking, support for multiple currencies, and smart features like voice-activated expense logging via Google Assistant, automated financial insights, and personalized spending recommendations.

The main contribution of the thesis is to develop an Android expense management application that is free and more convenient than existing applications on the market. The end result is a stable application with a user-friendly interface, providing an effective and interactive financial management experience. This application not only meets personal financial needs but also opens up opportunities for further feature development, such as integration with AI for user support or predictive budgeting tools in the future.

Student

(Signature and full name)

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Abriviation	Full Expression
BII	Ý định tích hợp (Built-in intents)
MVI	Mô hình kiến trúc MVI (Model-View-Intent)
MVVM	Mô hình kiến trúc MVVM (Model-View-ViewModel)
SOLID	5 nguyên lý thiết kế hướng đối tượng: SRP, OCP, LSP, ISP, DIP (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion)
UI/UX	Giao diện người dùng và Trải nghiệm người dùng (User Interface / User Experience)
URI	Định danh tài nguyên thống nhất (Uniform Resource Identifier)
XML	Ngôn ngữ đánh dấu mở rộng (eXtensible Markup Language)

CHAPTER 1. INTRODUCTION

1.1 Motivation

In the world of today, smartphones are very common and are used for many tasks in daily life because they are produced in large numbers and with great efficiency. And, one of the important tasks is managing personal money when they want to follow how is your money inflow, outflow, and it must easy to use, and some people want use as little manipulation as possible. However, many expense tracking apps for Android phones do not meet user needs because these apps often have simple features but lack a good design and easy interaction.

The problem is that current apps for Android often only let users track spending or make budgets and if people want to use more functions, they must pay a amount of money for the premium version. But after they buy a premium version, they still use this application with poor interfaces and miss useful features like voice control. This can make it hard for users to manage money easily and enjoyably because sometimes users don't want to typing many details about a transaction.

It is important to solve this problem because more people use smartphones to plan their finances. If we have a better Android app with Google Assistant or converting voice to a transaction, it can make users happier and meet their needs. In addition, it can also help people understand their money better and make smarter choices with all decisions related to their money.

So, creating an expense tracker app for Android that fixes these problems is necessary to maintain the habit of recording daily expenses. The app should have a clear and easy-to-use design and new features such as voice commands with Google Assistant or using voice to record daily transactions. This app can change how people handle money on their phones and it can let people have fun with record daily transactions routine, then who knows it will set a new example for other apps.

1.2 Objectives and scope of the graduation thesis

The main objective of this thesis is to research some specific apps related to the issue mentioned in the introduction and this thesis will focus on expense tracking apps for Android phones and those apps should be friendly, easy to use and have a high user experience, which is what users want. After that, this thesis will compare and study these applications to find their strengths, weaknesses, and parts that can be better. In addition, this thesis will focus on improving the user experience of the behavior app when using the app to track money.

Based on the main aim, the scope of this thesis is to review financial applications on Google Play and identify problems with those applications such as poor design, lack of voice control, and limitation of features. After reviewing apps and identifying properties, this thesis will describe how to create and build a new app related with finance with key features like a simple-to-use interface, easy budget planning and viewing list transactions, support for various currencies, and new tools like voice commands with Google Assistant to improve the user experience.

In addition, the thesis will work to bring about big changes in user enjoyment, ease of use, and app performance compared to other apps. By solving the listed problems and adding new tools, the new expense management app aims to create a new standard for finance apps on Android phones, offering users a more enjoyable and satisfying way to manage their money.

1.3 Tentative solution

The suggested solution uses the Model-View-Intent (MVI) pattern. This is a new and popular way to build software apps for Android using MVI. So, app development will be divided into three parts: the Model, which manages data and app logic; the View, which shows the user interface related to UI/UX design; and the Intent, which handles user actions and sends them as clear instructions to update the state of the app.

In this plan, the MVI pattern will use throughout project in the development of an expense tracker management app for Android phones. With the Model part, it holds all the app's data and logic, like a single source of truth. It's never directly changed, but updated by creating new states. In addition, it will handle tasks such as tracking expenses, managing budgets and transactions, and saving data. The View part will include the app's screens, such as a screen to add expenses in a day by hand or by voice, check transactions, and view spending history. The Intent part will process user actions that like voice commands with Google Assistant and update the Model to keep the app running smoothly.

The main goal of this solution by using the MVI pattern is unidirectional data flow, separation of concerns, and immutability. It will make all packages in the project organized follow a pattern, help developer easier to read that means data flow in a single direction from the Model to the View and back as intentions. The separation of concerns is shown by distinct roles for model, view, and intention components. The Model manages the state, the View handles UI rendering, and the Intent captures and communicates user actions. Lastly, immutability ensures that the state of the model remains unchanged once set. This guarantees will eliminate

unexpected side effects, and the application state will be stable and reliable.

Overall, using the MVI pattern for the expense management app shows a focus on building a strong, easy-to-update, and user-friendly app. By carefully following the MVI ideas, the app aims to provide a smooth and pleasant way for Android phone users to manage their money, helping to improve the development of mobile apps.

1.4 Thesis organization

The remaining parts of this thesis report are arranged as follows:

Chapter 2. Requirement Survey and Analysis: This chapter will explore a detailed study of the needs for the expense management application on Android user. It will look at what users want, their preferences, and expectations, and check existing expense tracking apps to find their common features and problems. After that, this section will dive into system design analysis.

Chapter 3. Technologies Used: This chapter will explain the all the technical used to build the expense tracker application that cover the name of technology, how to this technology works, some tools and libraries, and techniques used in building, and testing the app.

Chapter 4. System design, Implementation, and Evaluation: This chapter will describe the system design process, building steps, and testing methods for the expense management app. It will include topics like the MVI pattern design, user interface creation, coding methods, testing approaches, and how well the app works.

Chapter 5. Solution and Contribution: This chapter will present the finished app, showing its main features, functions, and benefits. It will explain how the app meets user needs and fixes problems found in other apps, as well as its importance for expense management apps on Android phones.

Chapter 6. Conclusion and Future Work: The last chapter will summarize the results, lessons learned from the study, and ideas for future improvements. It will talk about the app's successes, limits, and areas to make better, and suggest new directions for research and development in this area.

CHAPTER 2. REQUIREMENT SURVEY AND ANALYSIS

This chapter gives an overview of the requirement for building the expense management app for Android phones by starting with a survey to check the current state of expense tracking apps and trying to understand what users want and like. After the survey, a list of feature and key features will be present and then explain the detail on each feature to show how they work together in the application. Finally, non-functional needs are covered, pointing out important standards for performance, ease of use, and reliability to ensure the app provides a great user experience.

2.1 Status survey

A detailed study of the current state and needs for the expense management app was carried out, using three main sources: users, existing apps, and similar software. This section gives a clear analysis, comparison, and review of the strong points and weak points of current apps and studies in this area. It combines results from user surveys with a look at existing apps and similar software. The aim is to find and explain the important app features that must be built to fix current problems and improve the user experience.

2.1.1 User survey

To understand the needs and preferences of users, surveys and interviews were conducted with a diverse group of potential users. Key findings from these surveys include:

- **User Preferences:** Users want an application with an easy-to-use interface, quick operation, and the ability to use voice to control the application on demand. In addition, users need an application that runs smoothly, has few errors, and does not run in the background, causing battery drain. Users want to link the phone's virtual assistant with the application to add transactions when saying commands such as "I spent 50,000 on lunch today", view the transaction list "open transaction list", and open the application like that will save time and effort in data entry.
- **Common Complaints:** Users have to enter too much data for a transaction, users take a long time to complete a transaction, users have to take many steps to reach the desired function. The application has limited features and must pay for additional use.
- **Desired Features:** Based on user feedback, the necessary features include: managing transaction lists of each wallet, managing wallets, adding transac-

tions by input method or through virtual assistants, managing budgets for each category of each wallet, viewing overviews, statistics by graph.

2.1.2 Existing Systems

An analysis of existing expense tracker applications for Android reveals several common features and limitations:

- **Basic Functionality:** Most apps offer basic features like overview, wallet management, transaction management, budget management, adding transactions, debt management. Most of the features are too much to handle and do not have Google Assistant built in.
- **User Interface:** While some expense tracking apps have attractive designs, many have messy layouts and are hard to use. Users often feel annoyed with confusing navigation and features that are difficult to find.
- **Performance:** Many expense tracking apps have performance problems, such as slow loading, frequent crashes, and high battery use. Users also complain that apps take too long to save new expenses, which makes tracking money less convenient.

2.1.3 Similar Applications

A comparative analysis of similar expense tracking apps with over 1 million downloads was conducted to determine the strengths and weaknesses of each. The following table summarizes the comparison:

Application	Strengths	Weaknesses
MISA MoneyKeeper	Lots of features	Limited functionality, uninspired interface, slow and laggy app launch at first, no Google Assistance support.
Money Tracker	All the necessary features, no need to log in	Unfriendly interface, hard to find features, limited features, no Google Assistance support
Fast Budget	No need to log in, lots of information displayed on the home page.	Interface is too simple and unmodern, takes many steps to add transactions, limited features, no support for Google Assistance.
Money Lover	Modern interface, many functions, smooth, easy navigation	Takes many steps to add a transaction, limited functions, not supported with Google Assistant, high subscription cost

Table 2.1: Comparison of existing expense tracker applications

2.1.4 Key Features to Develop

Based on user analysis and feedback, the following key features are required for the proposed expense tracker app:

- **Enhanced User Interface:** A clean, intuitive, and visually appealing interface that ensures ease of use and quick navigation.
- **Add transactions with virtual assistant:** Add transactions automatically after speaking to virtual assistant to reduce data entry time. Optimize the number of steps required to complete a task.
- **Virtual assistant integration:** Allow users to interact with virtual assistant to increase app interactivity for enhanced user experience.
- **Unlock limits:** Unlimited wallet and budget creation.
- **Information storage:** Store data in internal memory, no internet connection required.
- **Performance Optimization:** Ensure stable performance, fast loading time and low battery consumption to enhance the overall user experience.

The proposed expense tracker app, by targeting existing limitations and integrating essential features, seeks to deliver an enhanced user experience and establish a new benchmark in the financial domain on Android smartphones.

2.2 Functional Overview

2.2.1 General use case diagram

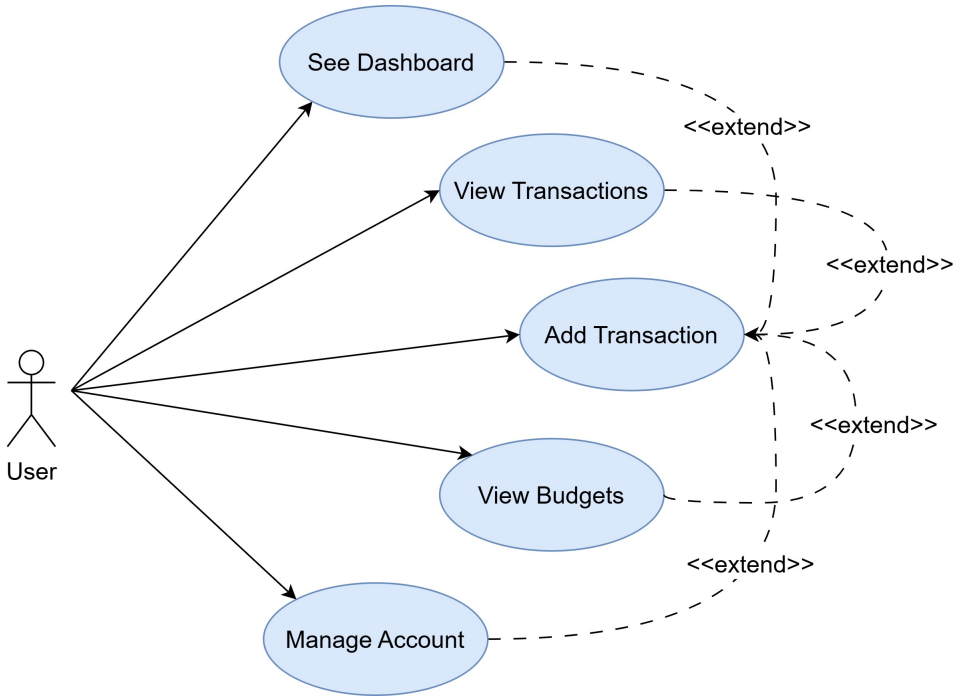

Figure 2.1: General use case diagram

2.2.2 Actor

- User
  - **Role**: Interact with the app to perform functions, tasks.
  - **Responsibilities**: Add transactions, manage transactions, manage budgets, manage debts, view overviews, manage wallets.

2.2.3 See Dashboard use case

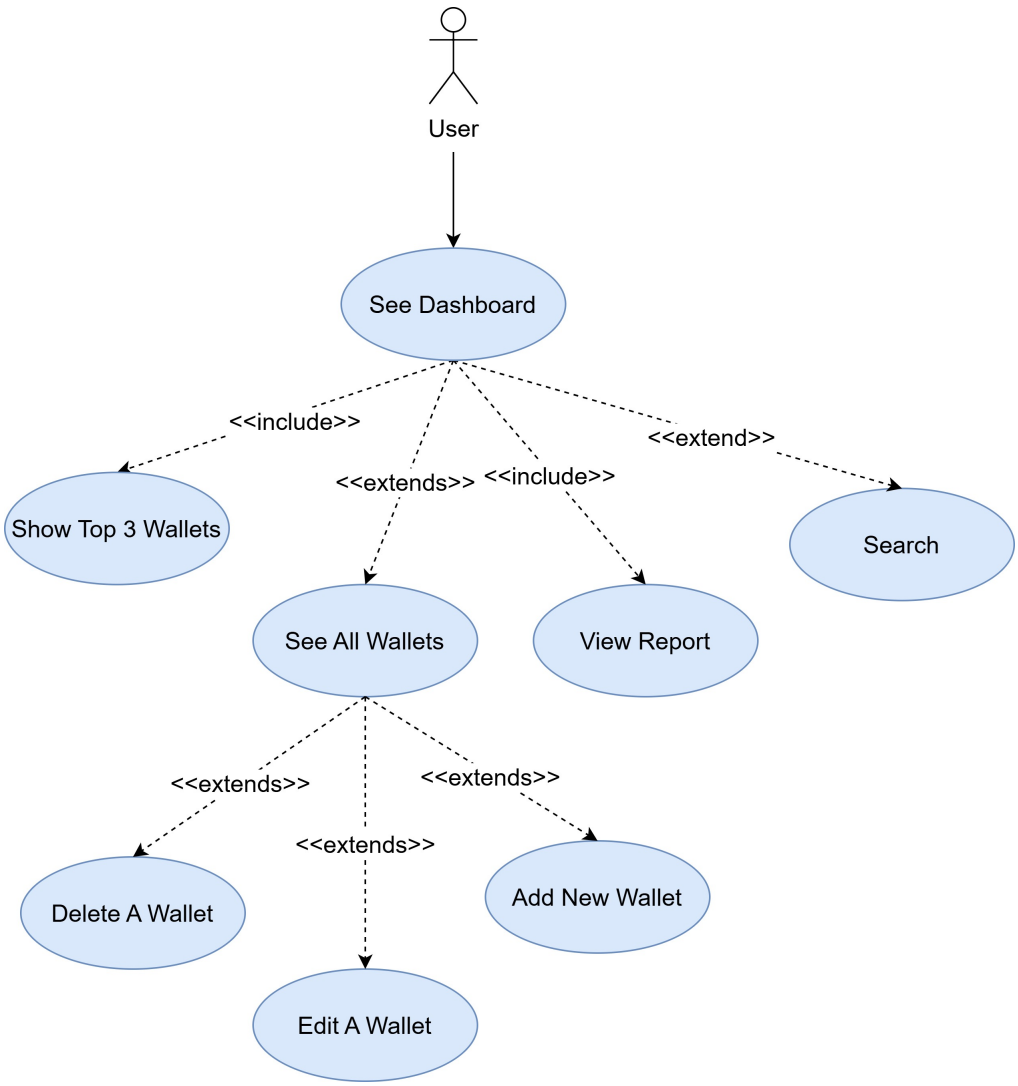

Figure 2.2: See Dashboard use case diagram

This is a detailed breakdown of the "See Dashboard" function in the application with the main use case being "See Dashboard" - where the user starts interacting with the system to quickly grasp the user's financial information.

- Related Use Cases
  - **Show Top 3 Wallets**: Automatically displays up to 3 most frequently or recently used wallets. This is a function that is always performed when opening Overview.

– **See All Wallets**: Allows users to view the entire list of wallets. Because users do not always open the list of all wallets. In which the extension of "See All Wallets" is Add New Wallet, Edit A Wallet, Delete A Wallet. With Use Case - Add New Wallet is used to add a new wallet while on the list screen. With Use Case - Edit A Wallet allows editing wallet information. With Use Case - Delete A Wallet is used to delete a wallet from the list.

– **View Report**: The summary financial report section (income/expenditure over time, charts, statistics). This is the main part of the Overview screen, always appearing on the Overview screen.

– **Search**: Feature allows searching for specific wallets and transactions. Not always necessary.

2.2.4 Transaction List use case

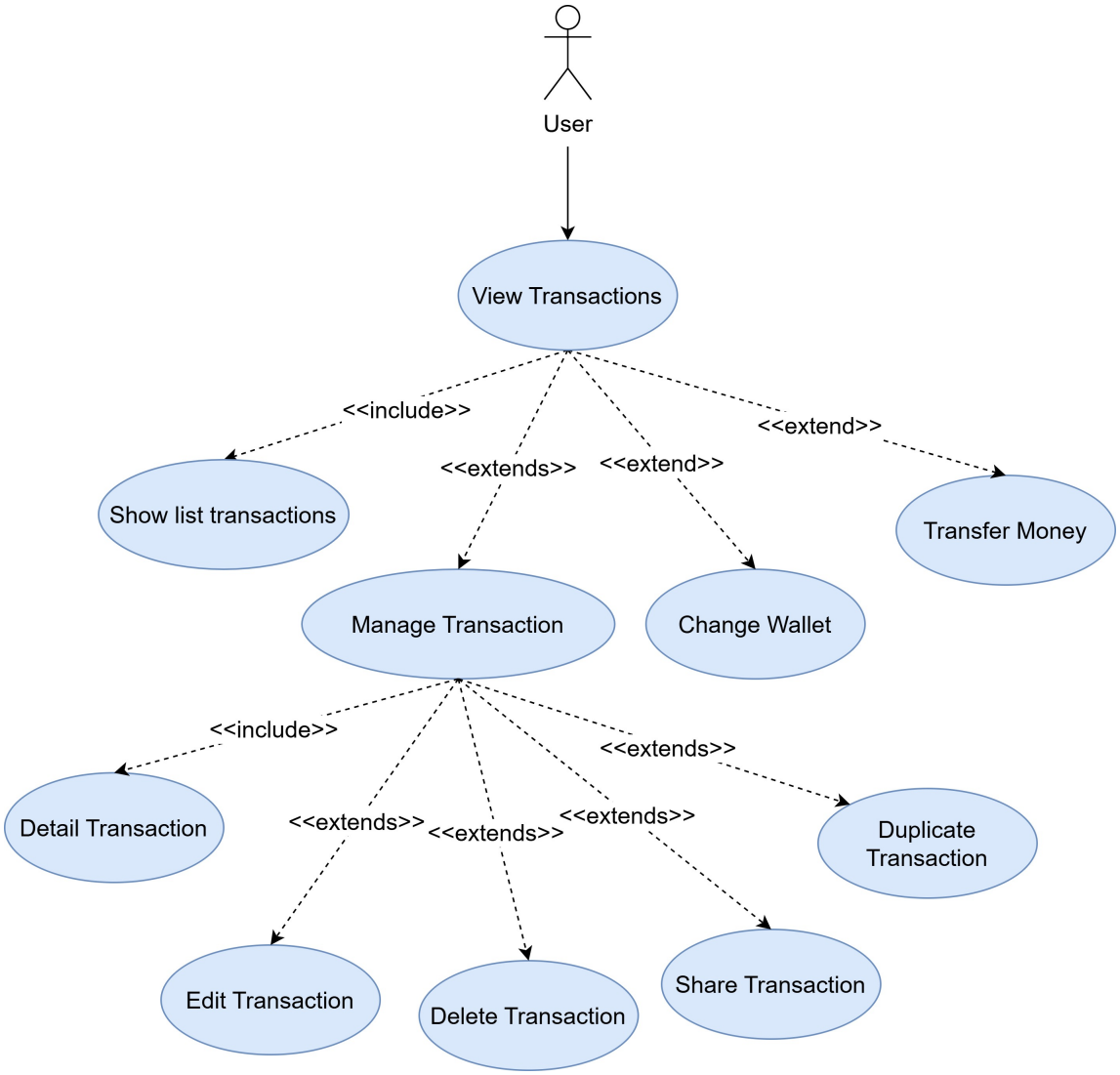

Figure 2.3: Transaction List use case diagram

With the Transaction List use case, this is where users can view, search, and manipulate the list of transactions such as spending, income, transfers, etc.

- Related Use Cases
  - **Show Transaction List**: When opening the Transaction List, the system always displays a list of transactions at all times along with the categories spent in an intuitive way, so this is a required behavior.
  - **Manage Transaction**: A collection of transaction management actions such as: viewing details, editing, deleting, sharing, duplicating transactions. Actions inside Transaction Management: Detail Transaction, Edit Transaction, Delete Transaction, Share Transaction:
    - **Detail Transaction**: Performed when the user clicks on any transaction and then opens a screen that allows editing, deleting, sharing, copying.
    - **Edit Transaction**: Allows users to update the transaction.
    - **Delete Transaction**: Deletes unnecessary transaction.
    - **Share Transaction**: Share a transaction with a screenshot of the transaction information via social networks, email...
    - **Duplicate Transaction**: Create a similar transaction (very useful for recurring transactions)
  - **Change Wallet**: Allows users to switch to another wallet to see the corresponding transaction for each wallet.
  - **Transfer Money**: Users can make transfers between wallets (from the Transaction List).

2.2.5 View Budget use case

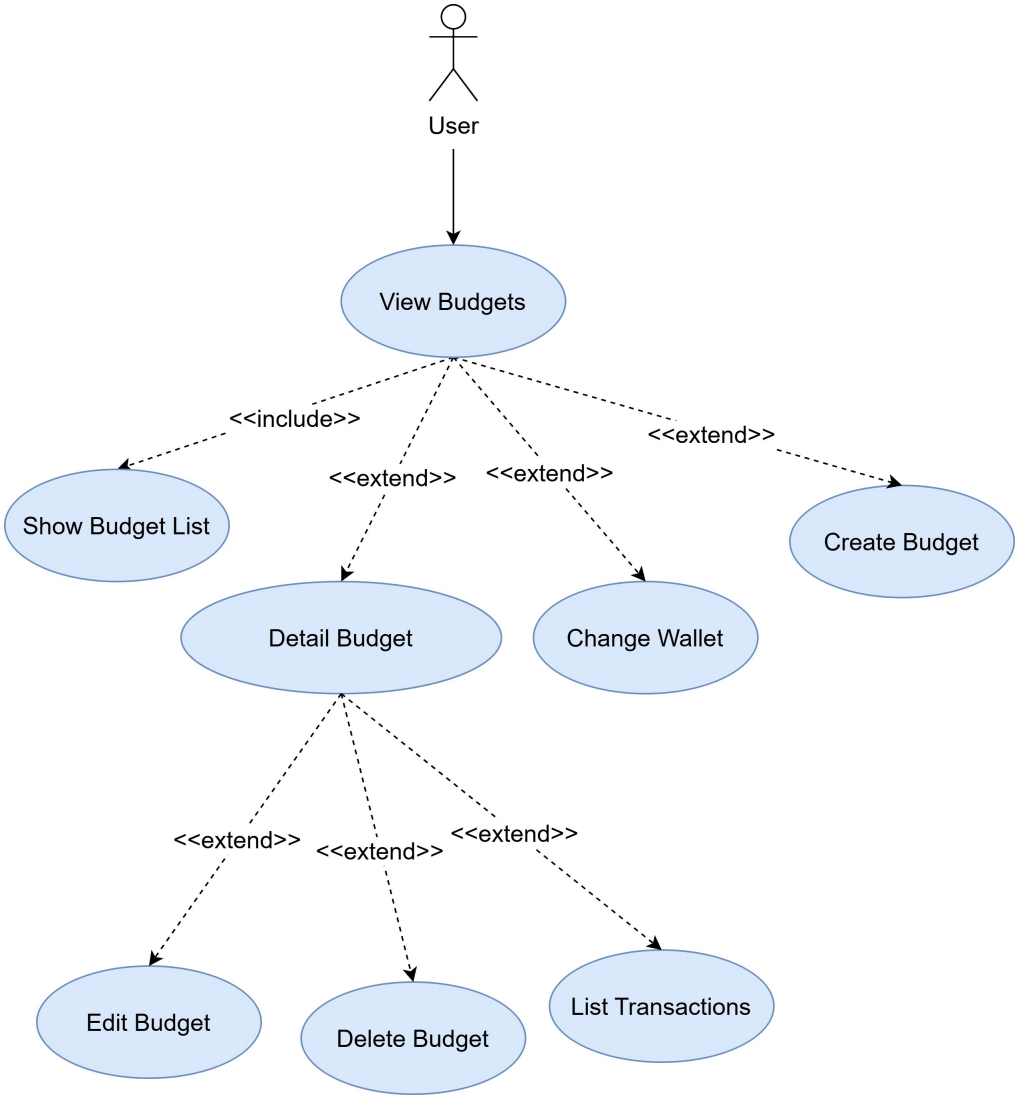

Figure 2.4: View Budget use case diagram

With the View Budget use case, this use case allows users to create different budgets for each category, edit each budget, delete budgets and finally track the budget to know how they are spending.

- Related Use Cases
  - **Show budget list**: Displays a list of existing budgets immediately after performing the Budget Management use case.
  - **Detail Budget**: View budget details, displayed after selecting a budget in the list of budgets. This is the extended center of specific budget processing behaviors.
    - **Edit Budget**: In the detailed view screen of a budget, this use case allows editing information about that budget.
    - **Delete Budget**: In the detailed view screen of a budget, this use case

allows deleting that budget.

* **List Transactions**: Displays transactions of a budget.

– **Change Wallet**: Change wallet applies to budgets because each wallet contains many different budgets.

– **Create Budget**: Allows users to create a new budget with the necessary information.

2.2.6 Add Transaction use case

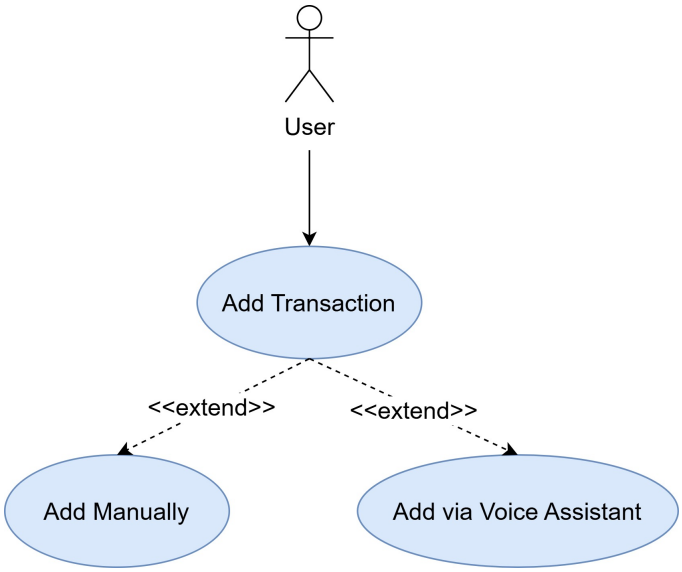

Figure 2.5: Add Transaction use case diagram

With the Add Transaction use case, this allows the user to add a transaction to the system in two different ways. This use case is activated when the user presses the add button on the main screen of the application or the user uses the phone's virtual assistant to give a command.

- Related Use Cases

  – **Add Manually**: Displays a screen to manually enter the transaction information.

  – **Add via Voice Assistant**: Allows the user to add a transaction via the Google Assistant virtual assistant by speaking or chatting with the assistant.

2.2.7 Account use case

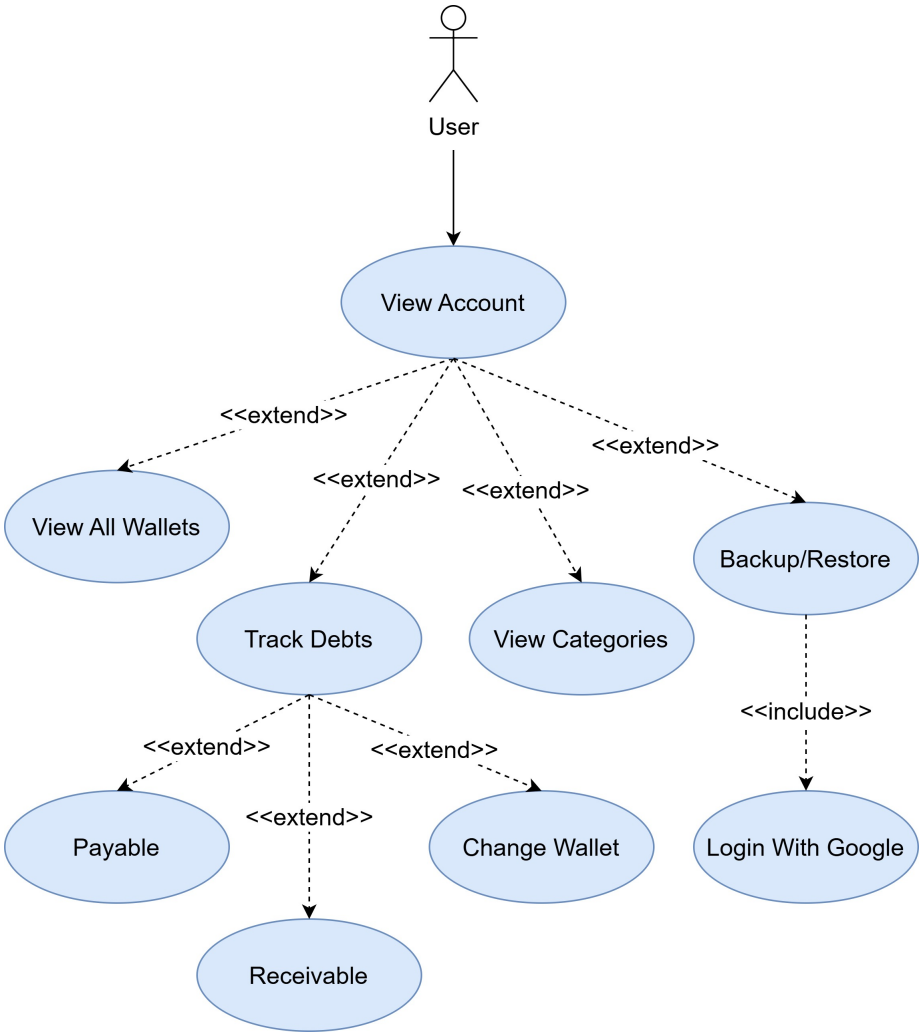

Figure 2.6: Account use case diagram

Use case Account provides users with the ability to manage information related to wallets, income/expenditure categories, and debts. Users can check existing wallets, current income/expenditure categories of the system to see if they are suitable for the user, and debts paid or unpaid.

- Related Use Cases
  - **All Wallets**: Allows users to track all existing wallets along with support for viewing total balances, classifying wallets and changing wallet information.
  - **Categories**: Users can define more or view existing spending categories such as: Food, Travel, Study... or income such as: Salary, Bonus,... The goal is to clearly classify transactions to make financial statistics and analysis easier.
  - **Debts**: Allows creating or tracking payable or receivable debts based on transactions with debt categories. Here users can make detailed notes,

mark the status of paid or not paid or paid in part. You can also transfer to another wallet to check the debt of each wallet.

– **Backup/Restore**: Supports users to backup and restore account data when users need to change devices or reinstall applications. To use this function, it is required to log in to a Google account.

2.2.8 Business process

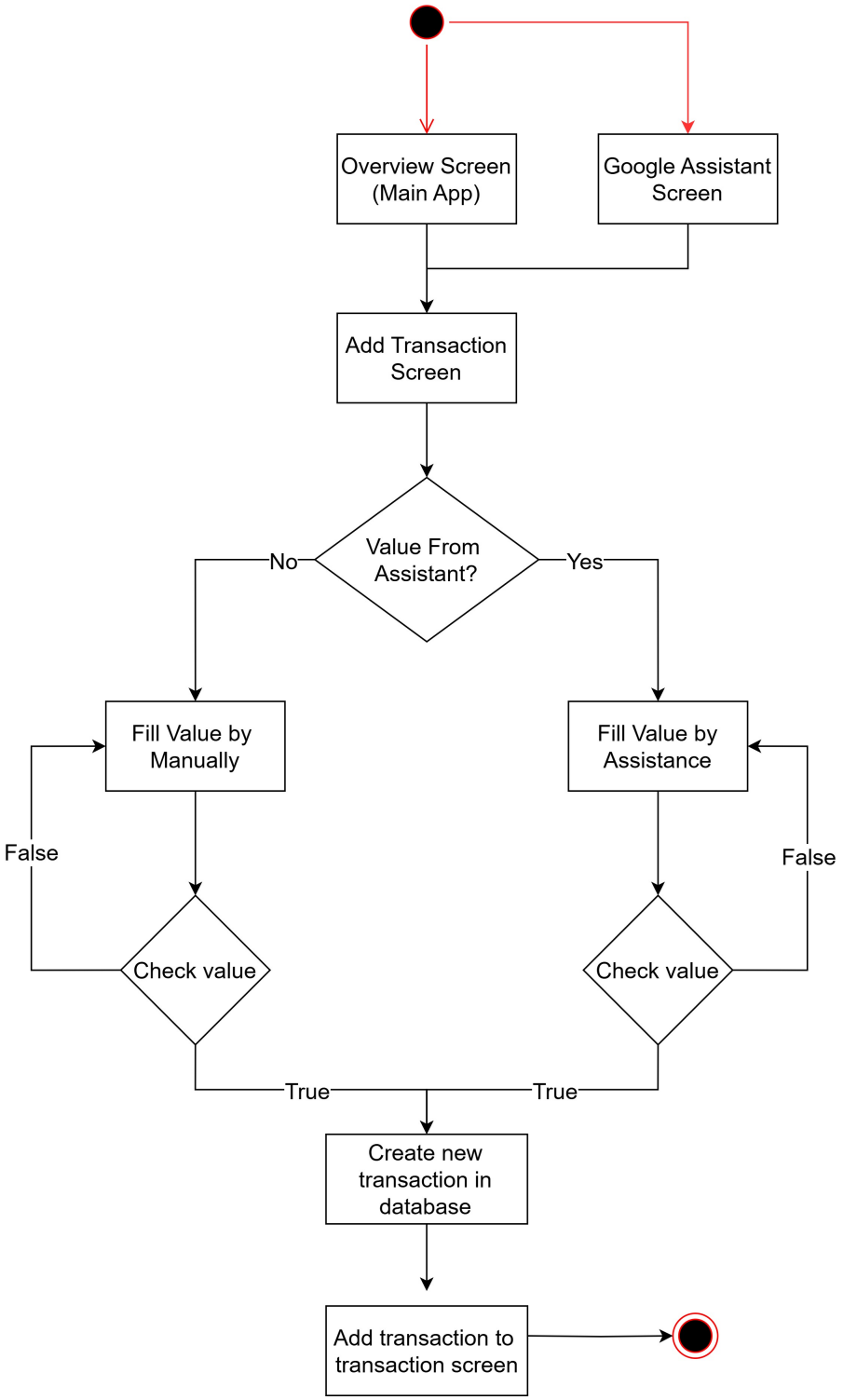

Figure 2.7: Add transaction business process diagram

Above is a flowchart describing the process of adding transactions in the application. The process is performed in order of steps. The beginning of the process is the black dot symbol representing the time when the user interacts with the application. If the user is on the main screen of the application, then the user will access the screen to add a new transaction, or if the user is on the main screen of the phone or anywhere and calls the Google virtual assistant, then both places will direct to the Add transaction screen. The next step is to check if the add transaction screen comes from the Virtual Assistant or not? If not, the user will fill in the values they want, if it comes from the Virtual Assistant, it will automatically fill in the values in the corresponding blank boxes. Next is to check if those values are valid or not? If not, ask the user to edit, but if they are valid, save to the database and display the new transaction on the transaction list screen. End of the process.

2.3 Functional description

2.3.1 Description of use case Overview

Use Case Name	See Dashboard
Description	The Dashboard screen is the main entry point after the user starts the spending management application. Here, the user can quickly view top wallets, access financial reports, search for information, and perform wallet management operations such as adding, editing, or deleting.
Trigger Event	The user opens the application and successfully launches the application.
Actor	User
Pre-conditions	The user has successfully logged into the system and the system has loaded the necessary wallet data and reports from the database.

Main Flow	1. The user launches the application and is navigated to the Overview screen. 2. The system displays the top 3 wallets with the highest balance. 3. The system provides the option to view all wallets. 4. The system provides the option to view financial reports. 5. The system provides a search box to retrieve transactions or wallets by keyword.
Alternative Flow	• **See All Wallets:** If the user selects "See All Wallets" then go to the screen listing all wallets. • **Add New Wallet:** If the user selects "Add New Wallet" in screen then open the screen to create a new wallet. • **Edit A Wallet:** If the user selects "Edit A Wallet" then open the interface to edit the selected wallet information. • **Delete A Wallet:** If the user selects "Delete A Wallet" then display a confirmation dialog and proceed to delete the wallet if agreed. • **Financial Report:** If the user selects "Financial Report" then display financial reports over time. • **Search:** When user uses search box, the system searches and displays appropriate results.
Post-conditions	The user is redirected to the desired feature and the corresponding data (wallet, transaction, report) is displayed according to the selection.

Table 2.2: Use Case Specification: Overview

2.3.2 Description of use case Transaction List

Use Case Name	Transaction list
Description	This function allows the user to view the transactions that have been added, from here the user can track the spending history, verify the receipts and payments.

Trigger Event	The user accesses the "Transaction List" screen from the main interface or after successfully adding a new transaction.
Actor	User
Pre-conditions	• The user has successfully started the application. • The system has at least one transaction recorded in the database.
Main Flow	1. The user selects the "Transaction List" item from the main screen or after creating a transaction. 2. The system retrieves all related transactions from the database. 3. The system sorts the transactions by time (default from new to old). 4. The system displays a list of transactions including: date, amount, transaction type (receipt/payment), description and related wallet. 5. The user can scroll to see more transactions. 6. Users can click on each transaction to view details or edit.
Alternative Flow	• **No transaction:** If no transaction is recorded, the system displays the message "No transaction yet". • **Click to a transaction:** If the user selects a specific transaction, the system navigates to the transaction detail screen.
Post-conditions	• The list of transactions is displayed to the user. • The user can perform further actions such as editing or adding new transactions.

Table 2.3: Use Case Specification: Transaction List

2.3.3 Description of use case Add Transaction

Use Case Name	Add Transaction
Description	This usecase allows the user to add a transaction to the system and that transaction can be entered manually or through the Google Assistant virtual assistant.
Trigger Event	The user selects “Add Transaction” from the main screen or via the Google Assistant.
Actor	User
Pre-conditions	• The system has at least one wallet created. • A stable network connection (if using Google Assistant).
Main Flow	1. The user selects “Add Transaction” from the main screen or from the Google Assistant. 2. The system navigates to the “Add Transaction” screen. 3. The system determines the input data source. 4. If from Google Assistant: Automatically fill in the recognized information (value, description, time...). 5. If not: The user manually enters the information fields. 6. The user checks and edits (if necessary) the pre-filled information or enters new information. 7. The system validates the transaction information (e.g., valid amount, selected wallet exists, etc.). 8. If valid, the system records the transaction in the database. 9. The system displays the transaction list again with the newly added item.

Alternative Flow	• **No value received:** If no value is received from the Assistant, the user is asked to manually enter the information. • **Invalid input:** If the input data is invalid (e.g. empty or negative amount), the system displays an error message and asks the user to correct it. • **Cancel operation:** If the user cancels the operation, the user returns to the previous screen without saving the data. • **Command not understood:** If Google Assistant does not understand the command, it suggests the user to enter it manually.
Post-conditions	• The new transaction is successfully saved to the database. • The user is redirected to the Transaction List screen with the newly created transaction.

Table 2.4: Use Case Specification: Add Transaction

2.3.4 Description of use case Budget Management

Use Case Name	Budget Management
Description	This usecase allows users to set up and track spending budgets by category for a specific period of time, helping users control spending more effectively.
Trigger Event	User accesses the budget management function from the main screen.
Actor	User
Pre-conditions	• User already has at least one wallet and spending category.

Main Flow	1. User accesses the "Budget Management" screen. 2. The system displays a list of current budgets (if any). 3. User selects "Add new budget" or edits an existing budget. 4. User enters information 5. The system checks the validity of the information (for example, the amount must be > 0). 6. If valid, the system saves the budget to the database. 7. The system updates and displays the budget list again. 8. During use, the system tracks spending by category and compares it with the budget
Alternative Flow	• **Invalid information:** If the budget information is invalid (for example, negative amount or missing category), the system displays an error message and asks the user to re-enter it. • **Cancel operation:** If the user cancels the budget creation/editing operation, the system returns to the budget list without changing the data.
Post-conditions	• The budget is successfully created or updated in the system. • The system starts tracking spending against the set budget.

Table 2.5: Use Case Specification: Budget Management

2.3.5 Description of use case Account

Use Case Name	Account
Description	This use case allows users to check information related to wallets, portfolios, debts. In addition, users can backup and restore data.
Trigger Event	The user accesses the account function from the main screen.

Actor	User
Pre-conditions	• The user has successfully started the app • The device has a network connection and a Google account.
Main Flow	1. The user navigate to the “Account” screen. 2. The system displays functions such as view wallet, view category, view debt. 3. The user selects the desired function. 4. The system navigates to the requested function.
Alternative Flow	• **Navigation error:** Restart the application if not navigated to the account function.
Post-conditions	• The system successfully navigates to the account function. • The user can select the desired functions.

Table 2.6: Use Case Specification: Account

2.4 Non-functional requirement

The expense management application needs to fully meet the non-functional requirements to ensure the quality of user experience and system stability. To achieve that, performance is a key factor that helps the system to handle smoothly when many transactions are loaded at the same time, while ensuring the fast loading speed of transaction lists, financial reports and budget information must be calculated optimally, while still optimizing battery consumption on mobile devices.

Reliability is also very important because the application needs to ensure continuous access to user data, so it is necessary to minimize data loss or corruption to help the system operate stably without interruption. In terms of ease of use, the interface should be intuitive and user-friendly, allowing users to easily perform tasks such as adding transactions, tracking expenses, setting budgets, and generating reports without complicated instructions. To achieve this, the interface design should be beautiful, easy to see, and responsive across different screen sizes and devices.

Maintainability is also an important requirement; the application should be built on a clear source code, have a reasonable module structure, and follow good programming standards, making it convenient to upgrade, fix bugs, and develop new features. In terms of technology, the system should use a secure, efficient, and scalable database to store and retrieve large amounts of users' financial information.

In addition, the application should be developed using modern and popular technologies such as Kotlin for Android. Security needs to be ensured by applying secure authentication mechanisms, encrypting data, and preventing common vulnerabilities related to user databases.

In short, the application needs to be able to optimize performance, improve reliability, and be easy to use. When there is a lot of data, the application still needs to operate smoothly and can still be expanded with additional functions to make it easy to update the application later.

CHAPTER 3. TECHNOLOGIES USED

In the previous chapter, I conducted a thorough requirements survey and analysis, providing insights into the current state of expense tracking applications, the required functionalities, and the unmet needs that my application was designed to address. I researched the key features that would enhance user engagement and identified the key use cases that my application should address. This chapter explores the technological underpinnings of my expense tracking application. I will discuss the various technologies, frameworks, and tools used throughout the development process. Essential considerations such as the choice of platform, programming language, library, and development environment will be thoroughly examined. Additionally, I will address why certain technologies were chosen and how they help achieve the desired functionality and performance. Specifically, I will discuss the importance of the Android platform to my project, the implementation of the Model-View-Intent (MVI) architecture-based technology, the programming languages and libraries used, as well as the tools and environments that facilitate efficient development and testing. By the end of this chapter, you will have a comprehensive understanding of the technology setup that supported the development of my expense tracking application and how these technologies work together to create a reliable, user-centric product.

3.1 Android

Android is officially designed and supported by Google, and devices using the Android operating system will account for 71.38 percent by 2024 based on a report by [1]. This is an operating system that is favored by many people because of its ease of use, beautiful interface, support for many user segments from cheap to expensive, security and smoothness. For those reasons, this is still a future operating system that brings experience to many people, so I chose this platform to deploy a spending management application specifically for Android users. Moreover, it provides a powerful SDK, well integrated with Google services such as Google virtual assistant. Good support with Google virtual assistant helps all users to easily use the application after being guided easily. In addition, the Android operating system also allows deep access to the system and supports App Actions for interaction between people and applications via virtual assistants.

3.2 Kotlin

Kotlin is an advanced, statically-typed programming language that operates on the Java Virtual Machine (JVM) and maintains full compatibility with Java. Devel-

oped by JetBrains, Kotlin has experienced a remarkable increase in popularity over the past few years, particularly in the realm of Android application development. In 2017, Kotlin was distinguished by Google as a preferred language for creating Android applications, which substantially fueled its widespread acceptance within the developer community.

Kotlin presents various advantages that render it an outstanding option for Android application development. Foremost among these is its concise and expressive syntax, in contrast to Java, which considerably minimizes redundant code and enhances the readability of code. This characteristic simplifies the act of writing, maintaining, and troubleshooting code, thereby elevating developer productivity.

Additionally, Kotlin integrates several state-of-the-art language features, including null safety, extension functions, and coroutines. Null safety is essential for preventing null pointer exceptions, a frequent issue in Java, by differentiating between nullable and non-nullable types during compilation. Extension functions allow developers to add new functionality to existing classes without modifying their original source code, fostering a more modular and extensible codebase. Coroutines offer an elegant solution for handling asynchronous programming, simplifying the execution of tasks such as network requests, database operations, and other time-intensive processes, all without impeding the main thread.

In the development of my expense tracker application, Kotlin is employed to implement a diverse set of features and functionalities, leveraging its contemporary capabilities to improve the overall development workflow. The language's concise and expressive syntax assists in creating a streamlined and efficient codebase, while its compatibility with Java allows for smooth integration with existing Java libraries and frameworks.

When selecting Kotlin for this project, other alternatives like Java, Swift (for iOS), and Flutter (a cross-platform framework using the Dart language) were considered. However, Kotlin was chosen for its robust support for Android development, its modern language features that facilitate easier coding, and its growing community and ecosystem.

Implementing Kotlin in the development of my expense tracker application leads to improved code quality, increased productivity, and access to an extensive collection of libraries and tools specifically designed for Android development. This guarantees the application's robustness, maintainability, and ability to deliver a superb user experience.

3.3 Jetpack Compose

After choosing a programming language, I want to focus on interface technology. With today's interface, users tend to want to use friendly interfaces while programmers want the interface to be designed quickly, but if they want a beautiful interface, it sometimes takes a lot of time to design with code. In the past, when programming Android, we could use XML layout files, but now Google has released and encouraged programmers to switch to using Jetpack Compose for UI because it integrates well and easily with Kotlin and ViewModel.

The goal is to solve as mentioned in chapter 2, the application needs to have a friendly interface, fast response and easy maintenance, so Compose is chosen to build the entire UI in the direction of declaring functions, making it easy to develop, customize and most importantly, can reuse interface components.

The main benefits of using Jetpack Compose compared to XML are numerous. First of all, declarative, the programmer describes the UI according to the current data state instead of controlling the interface step by step like XML, which helps avoid logical errors and is easier to maintain. Good integration with ViewModel and State hoisting will make Compose easy to combine with MVI architectures to update the interface based on dynamic data. Compose will help increase development efficiency because the time to write UI is reduced thanks to clean Kotlin APIs, live previews, and no need to rebuild the entire application to see the results. The UI is flexible and modern because it supports Material 3, animation, dynamic theming, dark/light mode more easily than XML. Finally, Compose is fully compatible with Kotlin because it does not need to write separate XML - all UI is in Kotlin code, so it is easy to manage.

3.4 Room Database

After choosing the technology for the interface, I choose the technology for storing user and application data. Here I use Room to interact with Android's Database. The reason for choosing Room is because this is a library in Jetpack Components and is officially supported by Google, compatible with other libraries such as ViewModel, Hilt, Flow that I will explain later. Moreover, it is suitable for offline-first applications, which need a stable data storage place and ensure performance. Finally, it is easy to maintain and expand the schema when the application develops new features.

In chapter 2 (design analysis), to achieve the goal of the expense management application, it is necessary to store transaction data, wallet, budget, user account in a sustainable way, ensure integrity, and easy to access even when there is no net-

work connection, so Room is chosen as Object Relational Mapping to manipulate with SQLite, making it easy to query, update and synchronize local data.

The main benefits of Room are automatic generation of safe SQL code, good integration with Flow and Coroutine, and Migration support. Room automatically generates SQL code through annotations and Kotlin classes to avoid SQL runtime errors and ensure data type checking at compile time. Good integration with Flow and Coroutine along with ViewModel to automatically update UI when data changes in case of saving a new transaction is a typical example.

3.5 ViewModel

After choosing the technology for database, I will choose the technology for data management in the UI. That is to use ViewModel, which is an official component in Jetpack components and is tightly compatible with other Android libraries.

With the goal of solving in chapter 2, the application requires the interface to always display the correct data (transaction, wallet, budget...) and not lose data when the user rotates the screen, switches tabs or temporarily leaves the application. Therefore, using ViewModel will store temporary data of the UI, ensuring that the interface can restore the state without having to reload all data from Room or server.

The main benefit of ViewModel is to manage the UI lifecycle because ViewModel lives longer than Fragment/Activity, helping to preserve the state of UI data throughout the lifecycle. ViewModel separates the processing logic and interface, helping to reduce the dependency between UI and data processing, supporting a clear MVVM architecture, easy to maintain and test. Furthermore, ViewModel integrates with LiveData and Flow to efficiently observe data, whenever Room updates data, ViewModel will detect and notify UI to update again. ViewModel is compatible with Hilt because it allows ViewModel to be easily injected through Hilt, which helps manage dependencies neatly.

3.6 Other Android Jetpack Components

Android Jetpack refers to a collection of libraries, tools, and best practice guidelines developed by Google, aimed at assisting developers in the creation of high-quality Android applications in an efficient and straightforward manner. This suite encompasses a diverse array of components specifically crafted to address frequently encountered challenges arising during the Android app development process. Excluding Room, several pivotal Android Jetpack components pertinent to our discourse include:

- **Flow**: Flow is designed to solve the need for lightweight, optimized asynchronous data stream processing from Data store or Room because data such as transaction list, wallet balance, budget... changes continuously, need to be observed and responded automatically when there is a change. The main benefit of using Flow is automatic UI update, combined with ViewModel and easy to combine with Coroutine to run asynchronously.
- **Coroutine**: Coroutine is a library with synchronous and asynchronous processing mechanisms to help manage long-running tasks that can block the main thread, making the application slow. The goal when using Coroutine is to access the Room database, save transactions, ... without blocking the main thread. The reason for choosing it is that it is lighter than Thread, easy to combine with Flow, Room, Compose and ViewModel.
- **Data store**: Data Store is a data storage solution that allows storing key-value pairs. DataStore uses Kotlin coroutine and Flow to store data in an asynchronous, consistent and shareable way instead of Android's old Shared Preference. The goal is to store user and application configuration data such as onboarding status, dark mode or light mode interface. The reason for choosing is easy integration with Flow and ViewModel to update UI when data changes.
- **Navigation**: Navigation is a modern library that helps manage screen navigation flow systematically and safely. The goal is to solve the need for clear, easy-to-control navigation when there are multiple screens: transaction list, wallet, budget, statistics... Its main benefits are defining an intuitive navigation flow, supporting back stack, deep link to receive data from Google Assistant, fully supporting animation, in addition to increasing consistency and limiting errors when controlling with Fragment Transaction.
- **WorkManager**: WorkManager is the recommended solution for continuous work, needing to be scheduled instead of Service used to handle background jobs. The settlement goal helps schedule notifications for users to record transactions at the end of the day.

3.7 Libraries

3.7.1 Dagger Hilt

Dagger Hilt is a library for automatic dependency management based on the Dependency Injection principle of SOLID. With the goal of managing dependencies between ViewModel, UseCase, Repository and Database classes to ensure Modularity, making code easy to change, extend, and deploy during application

development. It also supports lifecycle scoping to minimize memory usage.

3.7.2 Lottie

Lottie is library by Airbnb for Android and other platforms that decodes JSON-exported Adobe After Effects animations via the Bodymovin plugin. It enables developers to integrate intricate animations effortlessly into their apps, improving UI and UX without affecting performance. Lottie animations are scalable, resolution-independent, and can be seamlessly incorporated into Android XML layouts or through code.

3.7.3 Material 3

Material 3 is a design standard supported by Google used on Jetpack Compose with styles such as font styles, icon styles, layout allocation sizes. In addition, it also provides modern UI components used in the interface such as TopAppBar, BottomAppBar, ... With the goal of ensuring the UI of the application is modern, consistent, friendly and intuitive.

3.7.4 Vico Chart

Vico is a third-party library on github that integrates all types of charts that can be used with Jetpack Compose. This library will solve the problem of viewing financial reports as charts of revenue/expenditure trends, distribution by transaction type, time.

CHAPTER 4. DESIGN, IMPLEMENTATION, AND EVALUATION

Chapter 3 offers a detailed examination of the fundamental technologies and platforms incorporated into the construction of expense tracking applications. This encompasses the exploration of the Android operating system as the primary target platform, along with comprehensive analyses of pertinent databases and storage solutions. Furthermore, the chapter delves into the utilization of Kotlin as the chosen programming language and highlights the significance of various Android Jetpack components pertinent to data management and user interface-related lifecycles. In addition, there is an exploration of libraries utilized for enhancing functionality, such as Hilt for effective dependency management and the Vico charting library for advanced data visualization. Subsequently, Chapter 4 will present a thorough exploration of the design and implementation specifics of the expense tracking application. This includes an in-depth review of architectural design principles, user interface aesthetics, and the execution of essential functionalities. The chapter will conclude with a comprehensive evaluation of the application's performance metrics, usability aspects, and feedback garnered from users.

4.1 Architecture design

4.1.1 Software Architecture Selection

In this section, I will introduce the software architecture I chose for my application, which focuses on the Model-View-Intent (MVI) pattern. In the following sections, I will explain the general concept as well as its benefits and how to apply it in the application development process. In addition, I will also apply Clean Architecture to help the application be deployed with a consistent data flow.

a, The MVI Architecture

The Model-View-Intent (MVI) architecture has recently gained popularity in the Android development community due to its approach of dividing the application into three main parts: Model, View, and Intent. This architecture has a unidirectional data flow and assigns distinct roles to each component, making it easier to understand the programming flow, development, and maintenance. Thus, MVI becomes a typical choice for developing scalable and maintainable Android applications. Here are the basic concepts of each component:

- **Model**: Holds all the data and logic of the application, like a single source of information. It is never changed directly, but updated by creating new states.
- **View**: Represents the user interface (UI) renderer, displaying the state of the

application to the user without handling business logic. It updates based on changes in the Model's state.

- **Intent**: Represents actions by the user or the application itself, such as clicking a button or entering text, all related to what the user wants to do in the application. The View captures these intentions and sends them to the Model, which then performs the actions (such as updating the state of the application)

b, Apply MVI into the Expense Tracker Application

In my application, I have adopted MVI architecture to represent the communication between data, user interface, and business logic. Here, I will present each component of MVI in my application:

The Model in my application includes the following data layers:

- **Core Model Classes**: This model represents the core data models needed for the project representing the main business entities. It is the foundation for the entire functionality of the application and it is independent of UI or technology. In my application, the core model is divided into 3 main parts: Domain Models, Data Layer Models and Relationship Models.
  - **Domain models**: Domain models serve as core business entities that represent and manage financial data in a structured and logical manner. Each domain model has a specific role. Currency data class represents currencies, storing information such as currency codes and symbols, which help identify the currency unit in the entire system. Category data class classifies income/expenditure into meaningful groups such as dining, transportation, or compensation, and supports hierarchy to help users organize finances visually. CategoryGroup data class organizes categories into larger parent groups, creating a clearer and more manageable classification structure. Transaction data class is the core data model for recording financial transactions such as spending or income, playing a central role in tracking users' financial activities. Wallet data class represents funds such as cash wallets, bank accounts, or e-wallets, helping to separate and manage separate financial resources. Budget data class allows users to set spending limits for each category within a specific period of time, thereby controlling personal finances and limiting overspending.
  - **Data Layer Entities**: In the Data Layer of the application, entities act as models that map directly to the database (Room) to store information persistently and support efficient retrieval. Each entity corresponds

to a domain model at the business layer but is designed to fit the constraints and structure of the storage system. CurrencyEntity is the database representation of the Currency model, which persists information about currency codes and symbols. CategoryEntity stores expense/income categories, which restores the classification structure when the user reopens the application. WalletEntity represents a wallet or account in the database, allowing users to manage multiple sources of funds accurately. TransactionEntity is the master record of financial transactions, ensuring that each income or expenditure is fully recorded and can be traced back to the past. BudgetEntity supports saving spending plans by category, helping to restore and track established budgets over time. Each entity contains mapping methods to easily move back and forth between the domain model (used in business logic) and the entity (used in storage), ensuring a clear separation between the business layer and the data layer.

  - **Relationship Models**: Relationship Models act as a connection between core entities to create aggregate data structures for displaying and processing more complex logic. A typical example is the WalletWithCurrency model, which is used to combine information about a wallet (Wallet) with the related currency (Currency). This model helps provide a unified overview of wallets and their respective currencies, which is useful in user interfaces where wallet information needs to be displayed in a complete and accurate manner.

- **MVI Integration**: The models are used in the MVI architecture through:
  - **State Classes**: Data classes implementing interface MviStateBase that represent UI state.
  - **Intent Classes**: Sealed interfaces implementing interface MviIntentBase that represent user actions.
  - **Event Classes**: Sealed interfaces implementing interface MviEventBase for one-time events.

Views consist of user interface components, primarily defined by activities and functions annotated as composable:

- **Activity**: this is considered the starting point for screens and each activity will contain multiple composable layout files for each screen. Examples include:
  - **Splash Activity**: Displays the app logo and initializes app loading.
  - **Onboarding Activity**: Provides detailed instructions to assist users in

completing the initial setup process.

  - **Main Activity**: This is where all screens are displayed as well as navigation between each screen such as overview view, transaction list, add transaction, budget list,... This is also considered the center of the application.

- **Composable Layout Files**: Contains layout files for each screen in the application. It also contains files for each component that makes up a screen, but is not mentioned here. These layout compose files are in an activity and do not contain other calculation logic.
  - **HomeScreen**: Main dashboard showing account overview and recent transactions.
  - **TransactionScreen**: List and manage all financial transactions.
  - **BudgetScreen**: Track spending against a predefined budget.
  - **AccountScreen**: Manage and set up user accounts.
  - **AddTransactionScreen**: Create new expense or income entries.
  - **EditTransactionScreen**: Modify existing transactions.
  - **DetailTransactionScreen**: Display comprehensive transaction information.
  - **MyWalletsScreen**: Manage all user wallets/accounts.
  - **AddWalletScreen**: Create a new wallet.
  - **TransferMoneyScreen**: Transfer money between wallets.
  - **ChooseWalletScreen**: Wallet selection screen.
  - **AddBudgetScreen**: Create spending limits for categories.
  - **CategoryScreen**: Category selection and management.
  - **CurrencyScreen**: Select and provide information regarding currencies.
  - **EnterAmountScreen**: Dedicated screen for entering amounts.

Intent is a class or object that represents the user's intention. When the user interacts with the interface, the View will create an Intent and send it to the ViewModel for processing, and then the ViewModel updates the state of the Model. So below I will talk about where the Intent sends the event to update the View, which is the ViewModel, and consider the Intent as a bridge to process.

- **ViewModel Classes**: The ViewModel will contain 3 parts including: Intent

processing, the result of Model handling, and finally the states of the View. Here, there are 2 parts that will be related to each other, which are the result of Model processing and the state of the View because that is how the View-Model updates the View. Each screen will have a different ViewModel, and the ViewModel will be considered the processing center.

- **HomeScreenViewModel**: Used to manage the Overview screen with main functions such as creating a default wallet or loading and displaying wallets, and setting up data to create statistical graphs.
- **AddTransactionViewModel**: Handles tasks related to creating new financial transactions on the Add Transaction screen.
- **EditTransactionViewModel**: Manages editing of existing transactions of the edit transaction screen.
- **TransactionViewModel**: Manages all information related to the Transaction Screen and focuses on displaying transaction history.
- **DetailTransactionViewModel**: Manages the detailed view of the transaction.
- **BudgetViewModel**: Manages budget display and tracking.
- **AddBudgetViewModel**: Handles the creation of new spending budgets.
- **MyWalletsViewModel**: Manages wallet listing and operations.
- **AddWalletViewModel**: Handles the creation of new wallets.
- **TransferMoneyViewModel**: Manages transfers between wallets.
- **ChooseWalletViewModel**: Manages wallet selection for other screens.
- **CategoryViewModel**: Manages category browsing and selection.
- **CurrencyViewModel**: Manages currency selection.

c, Enhancements to MVI Architecture

I have refined the fundamental MVI architecture with various enhancements to more effectively meet the demands of my expense tracker application.

- **Enhanced MVI Base Architecture with Modern Reactive Streams**: Upgrade from traditional LiveData to StateFlow/SharedFlow for better coroutine integration and more powerful reactive programming capabilities.
- **Type-Safe MVI Contract System**: Implementing a strict contract-first approach with sealed interfaces for type-safe Intent and Event handling, ensuring

compile-time safety and preventing invalid state transitions.

- **Hybrid Feature and Clean Architecture Pattern**: Combined feature with clean architecture layers within each feature, creating a scalable modular structure.
- **Advanced Screen-Level Component**: Each screen is treated as a micro-module with its own contract, content, and components directory, promoting high cohesion and low coupling.
- **Multi-Scoped Dependency Injection Architecture**: Implemented differentiated DI scoping with ViewModelComponent: Use Cases (short-lived, per ViewModel) and SingletonComponent: Repositories, Data Sources (app-lifetime)
- **Feature Modularization**: Hierarchical module aggregation where each feature exposes a single module that internally aggregates all its dependencies, simplifying app-level configuration.

4.1.2 Overall design

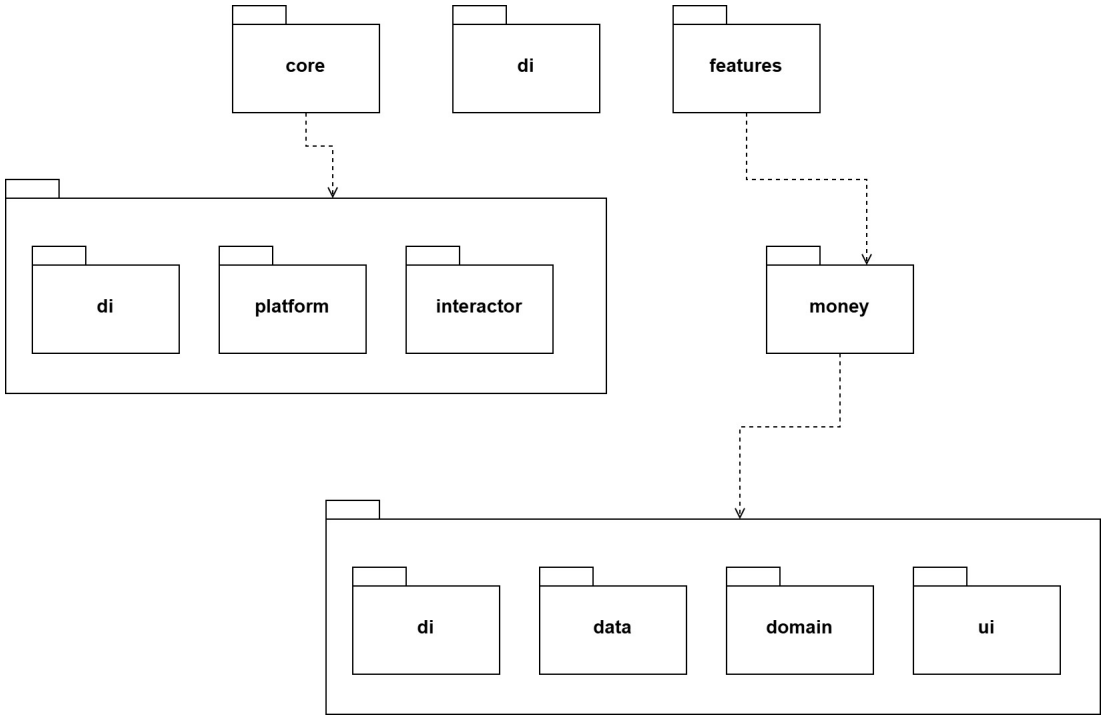

Figure 4.1: Package dependency diagram

4.1.3 Detailed package design

Add transaction package design

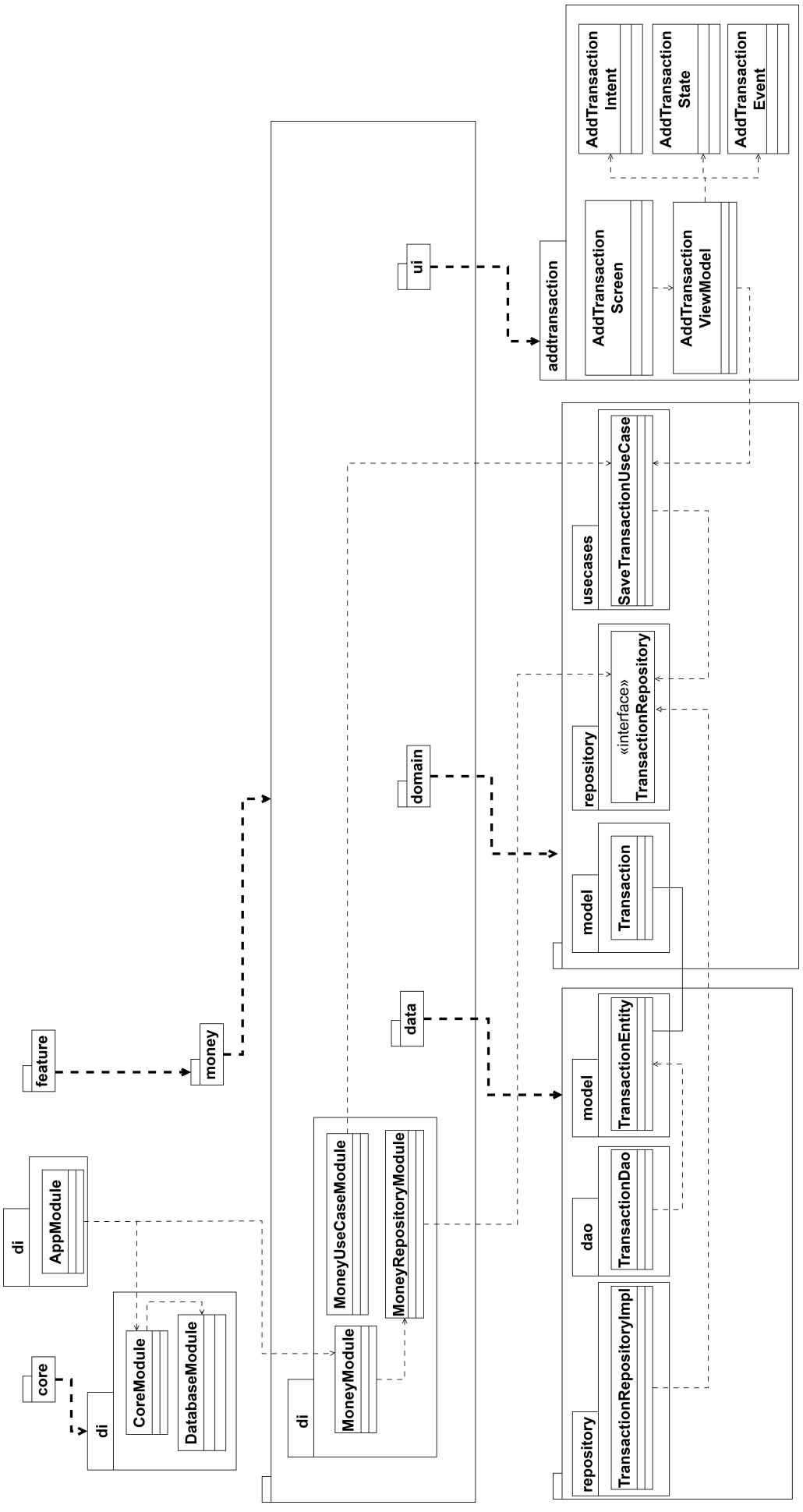

Figure 4.2: Add Transaction - Detailed Package Design

This is a flow to perform additional transactions, the package structure separates the layers following Clean Architecture and uses Hilt to manage dependencies.

Main structure layer:

- **di/**: Includes Hilt modules (AppModule, CoreModule, DatabaseModule, MoneyModule, MoneyUseCaseModule, MoneyRepositoryModule) to provide instances for classes (Repository, UseCase) through Hilt.
- **data/**: Contains data-related components: `dao/TransactionDao, model/TransactionEntity, repository/TransactionRepositoryImpl.TransactionRepositoryImpl` depends on `TransactionDao` and maps `TransactionEntity` to Transaction domain. Helps the domain not depend on the specific implementation details of the `/data` layer.
- **domain/**: Is the pure business logic layer: `model/Transaction` - domain object, `repository/TransactionRepository` - a class interface that provides API to get data from /data layer, `usecases/SaveTransactionUseCase` - SaveTransactionUseCase depends on TransactionRepository to perform transaction saving logic.
- **ui/addtransaction/**: `AddTransactionViewModel` depends on `SaveTransactionUseCase` to handle logic when user saves transaction. `AddTransactionViewModel` depends on AddTransactionIntent / State / Event to manage Intents sent from UI, current State and Event emitted.

4.2 Detailed design

4.2.1 User Interface Design

The user interface design for the app is geared towards a variety of devices with different screen resolutions and sizes. The key considerations for the user interface design are as follows:

a, Target Device Specifications

- **Screen Resolution**: The application is designed for screens with resolutions between 720x1280 (HD) and 1440x2560 (QHD) pixels, ensuring it works on many contemporary smartphones and tablets.
- **Screen Size**: The design supports screen dimensions ranging from 5.1 to 6.3 inches, catering to devices with 16:9 and 16:10 ratios, thereby encompassing the majority of prevalent mobile devices available.
- **Color Depth**: The interface offers 16 million colors (24-bit color), guaranteeing vivid and high-quality visuals.

b, Design Standards and Conventions

To ensure a consistent and user-friendly interface, the following design principles and conventions are applied:

Button Design

- **Shape and Size**: Buttons feature rounded edges with a touch target of at least 48dp x 48dp to improve access and use.
- **Color**: Primary buttons employ a unique color (e.g., blue) for emphasis, whereas secondary buttons use a neutral color (e.g., green).
- **Text**: Button labels are short and utilize a sans-serif font for clear readability.

Control Elements

- **Input Fields**: Text input fields are distinguished by a light gray border and, when active, they expand to feature a highlighted border to draw user attention. Placeholder text is provided as a form of guidance to assist users in understanding what to input.
- **Switches and Checkboxes**: Toggle switches and checkboxes are crafted to facilitate easy interaction, offering a clear visual representation of their on and off states to the user.

Message Display

- **Position**: Error messages as well as feedback notifications are strategically positioned close to the relevant input fields to provide immediate clarification or at the screen's base for notifications of a broader context.
- **Style**: These messages employ colors that contrast with the background to stand out and remain concise while still being informative.

Color Scheme

- **Primary Colors**: The application utilizes a primary color scheme predominantly featuring shades of blue, which serves the dual purpose of reinforcing the brand's identity and highlighting essential actions.
- **Secondary Colors**: Colors such as black and white are employed as secondary shades, contributing to the emphasis of significant elements by ensuring their prominence.

By following these design principles meticulously, the application seeks to deliver a smooth and aesthetically pleasing user experience that is uniform across various devices and screen dimensions. These guidelines play a crucial role in pre-

serving uniformity, bolstering usability, and elevating the overall visual appeal of the application.

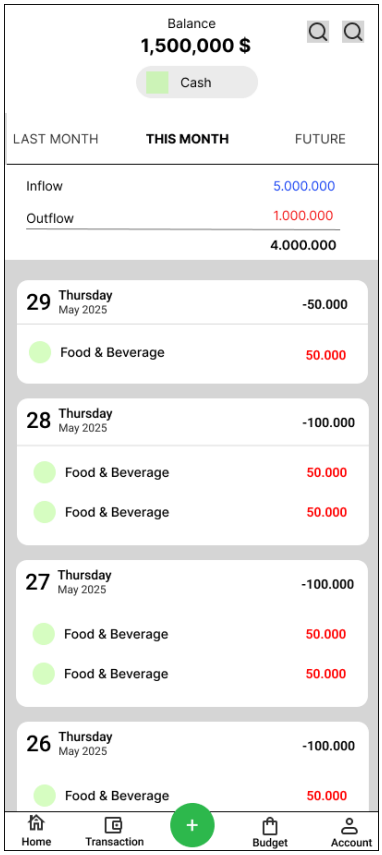

Figure 4.3: Interface design illustration Add Transaction Screen

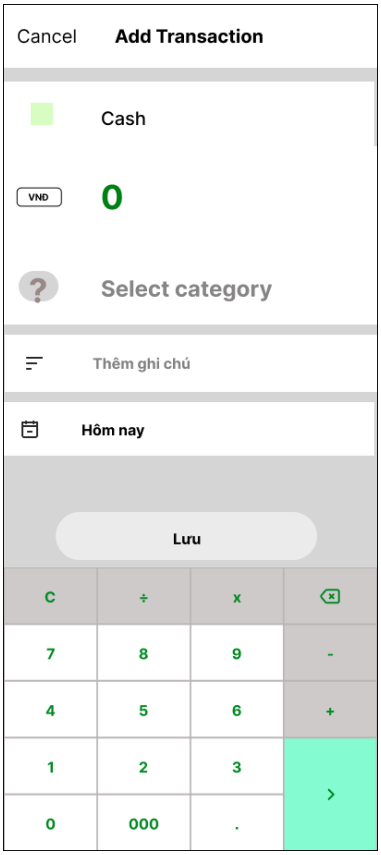

Figure 4.4: Interface design illustration Add Transaction Screen

4.2.2 Layer design

a, Layer design for add transaction

Property	Description
getWalletsUseCase	Use case for retrieving available wallets from the data layer.
saveTransaction UseCase	Use case for saving transaction data to the data layer.
viewState	Keeps the current state of the add transaction screen including current wallet, added amount, notes, categories,...

Table 4.1: Properties of the AddTransactionViewModel Class

Method	Description
loadWallets()	Loads available wallets using GetWalletsUseCase and sets the first wallet as default.
processIntent()	Main method that processes all user intents and delegates to appropriate handler methods.
saveTransaction()	Validate and save the transaction using SaveTransactionUseCase then determine the transaction type (in or out) based on the category. Manage the status of the transaction save as successful or failed.
handleNumpadPress()	Handles all numpad button presses including digits, operators, and special actions.

Table 4.2: Methods of the AddTransactionViewModel Class

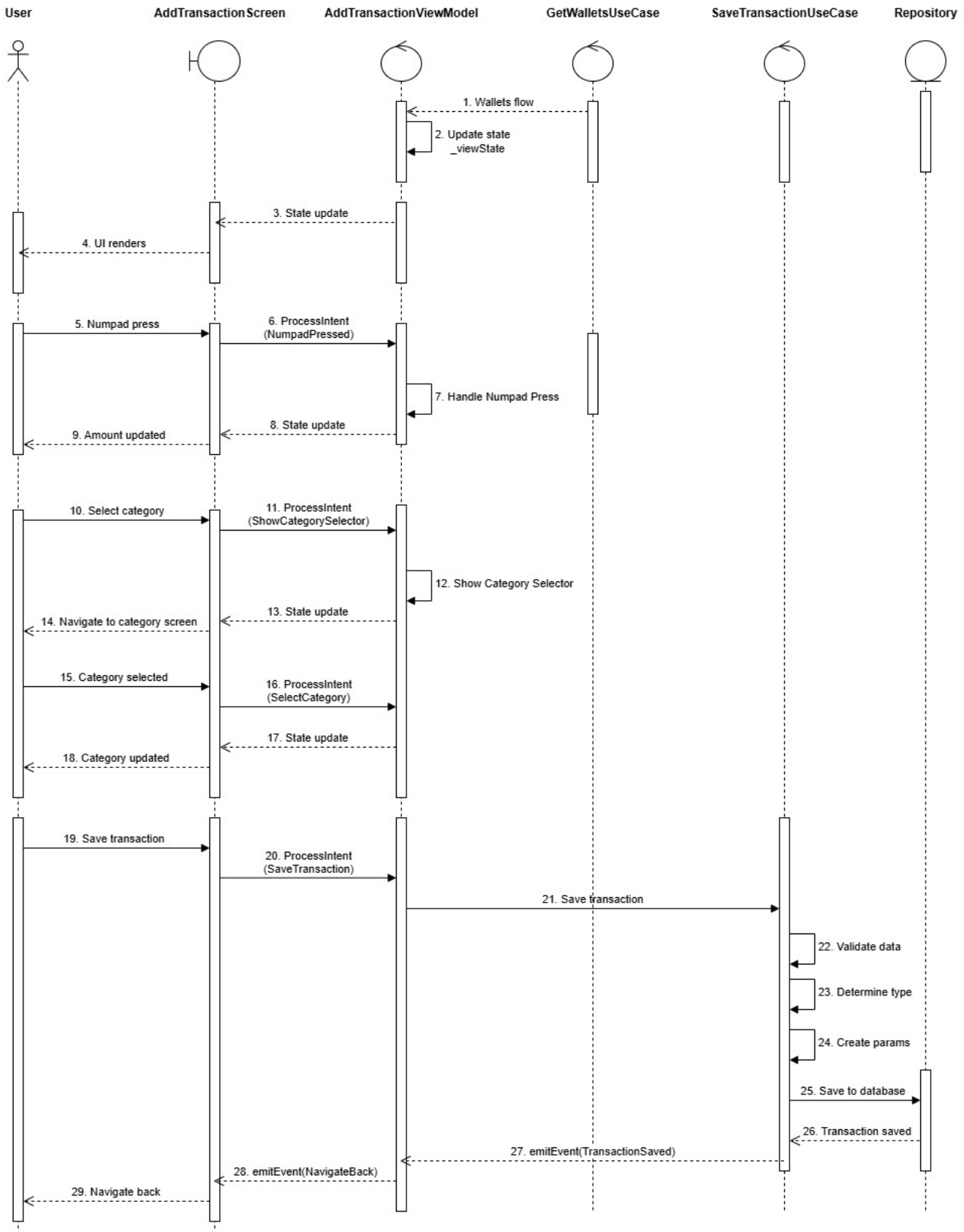

Figure 4.5: Add transaction sequence diagram

b, Layer design for list transactions

Attributes and Methods of TransactionsScreenViewModel	
Attribute	**Description**
getTransactions UseCase	Use case for fetching transactions data.
getTransactionBy IdUseCase	Use case for retrieving a specific transaction by its ID.
observeTransactions UseCase	Use case for observing real-time transaction updates.
observeWalletBy IdUseCase	Use case for observing real-time wallet updates.
currentWalletId	Stores the ID of the currently selected wallet.
viewState	Overridden state flow that holds the current UI state of Transaction Screen.
Methods	**Description**
processIntent(intent: TransactionIntent)	Overridden method that processes user intents.
selectWallet()	Switches to a different wallet or total wallet view. Handles logic for both individual wallets and the "All Wallets" aggregated view. Updates current wallet ID and triggers appropriate data loading..
loadTransactions()	Loads transactions for the currently selected individual wallet. Uses `ObserveTransactionsUseCase` with wallet-specific parameters. Handles loading state, error handling, and updates UI state with grouped transactions, inflow, and outflow calculations.
loadAllWallets Transactions()	Loads transactions from all wallets combined. Similar to loadTransactions() but uses `ObserveTransactionsUseCase.Params.AllWallets` and calculates total balance from all transactions.

Table 4.3: Attributes and Methods of TransactionsScreenViewModel

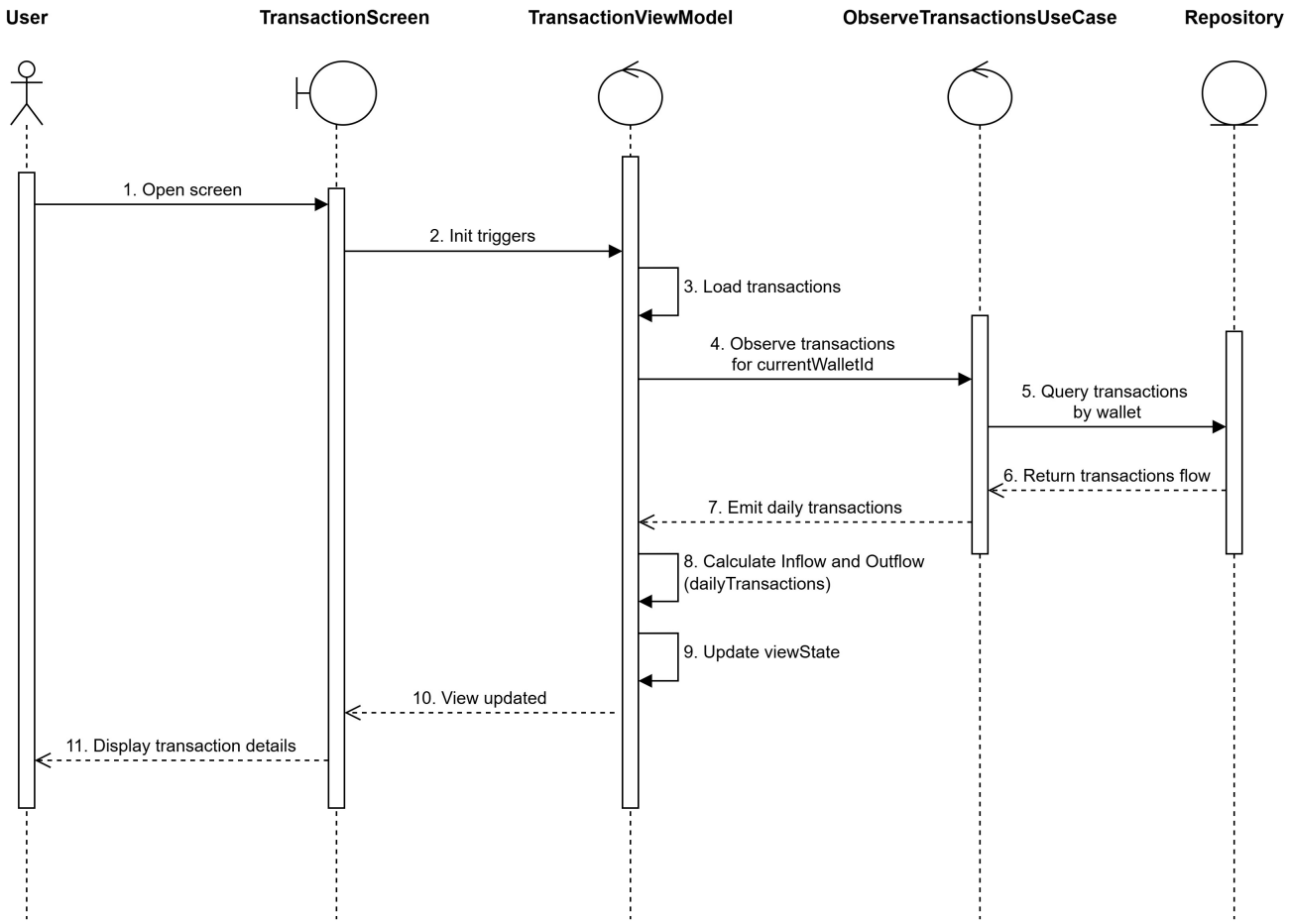

Figure 4.6: List transaction sequence diagram

4.2.3 Database design

Entity	Attributes and Description
WalletEntity	• **Attributes**: – *id*: Int (Primary Key) – *walletName*: String – *currentBalance*: Double – *currencyId*: Int (Foreign Key to CurrencyEntity) – *icon*: String – *isMainWallet*: Boolean • **Description**: Represents a financial container (wallet). Each wallet holds money in a specific currency and tracks its balance. Users can have multiple wallets for different purposes (e.g., savings, daily expenses, investments).
CurrencyEntity	• **Attributes**: – *id*: Int (Primary Key) – *currencyName*: String – *currencyCode*: String – *symbol*: String – *displayType*: String – *image*: ByteArray (nullable) • **Description**: Defines the monetary currencies with each currency contains display and formatting information to ensure proper money representation across different regions and cultures.
	Continued on next page

Continued from previous page

Entity	Attributes and Description
TransactionEntity	• **Attributes**: – *id*: Int (primary key) – *walletId*: Int (Foreign Key to WalletEntity) – *amount*: Double – *transactionType*: TransactionType (enum) – *transactionDate*: Date – *description*: String? (nullable) – *categoryId*: Int (Foreign Key to CategoryEntity) – *withPerson*: String? (nullable) – *eventName*: String? (nullable) – *hasReminder*: Boolean (defaults to false) – *excludeFromReport*: Boolean (defaults to false) – *photoUri*: String? (nullable) • **Description**: Represents a financial transaction. It serves as a comprehensive record of money movement, capturing both income and expenses across different wallets.
CategoryEntity	• **Attributes**: – *id*: Int (Primary Key, auto-generated) – *metaData*: String – *title*: String – *icon*: String – *type*: Int (1=INCOME, 2=EXPENSE, 3=DEBT_LOAN) – *parentName*: String? (nullable, for hierarchical categories) • **Description**: Organizes transactions into meaningful groups for better financial management and reporting. Supports a hierarchical category structure for detailed expense classification.

Continued on next page

Continued from previous page

Entity	Attributes and Description
BudgetEntity	• **Attributes**: – *budgetId*: Int (Primary Key, auto-generated) – *categoryId*: Int (Foreign Key to CategoryEntity) – *walletId*: Int (Foreign Key to WalletEntity) – *amount*: Double – *fromDate*: Date – *endDate*: Date • **Description**: Manages financial planning by setting spending limits for specific categories and wallets over defined time periods. Enables users to control and monitor their expenses against planned budgets.

Table 4.4: Database explanation

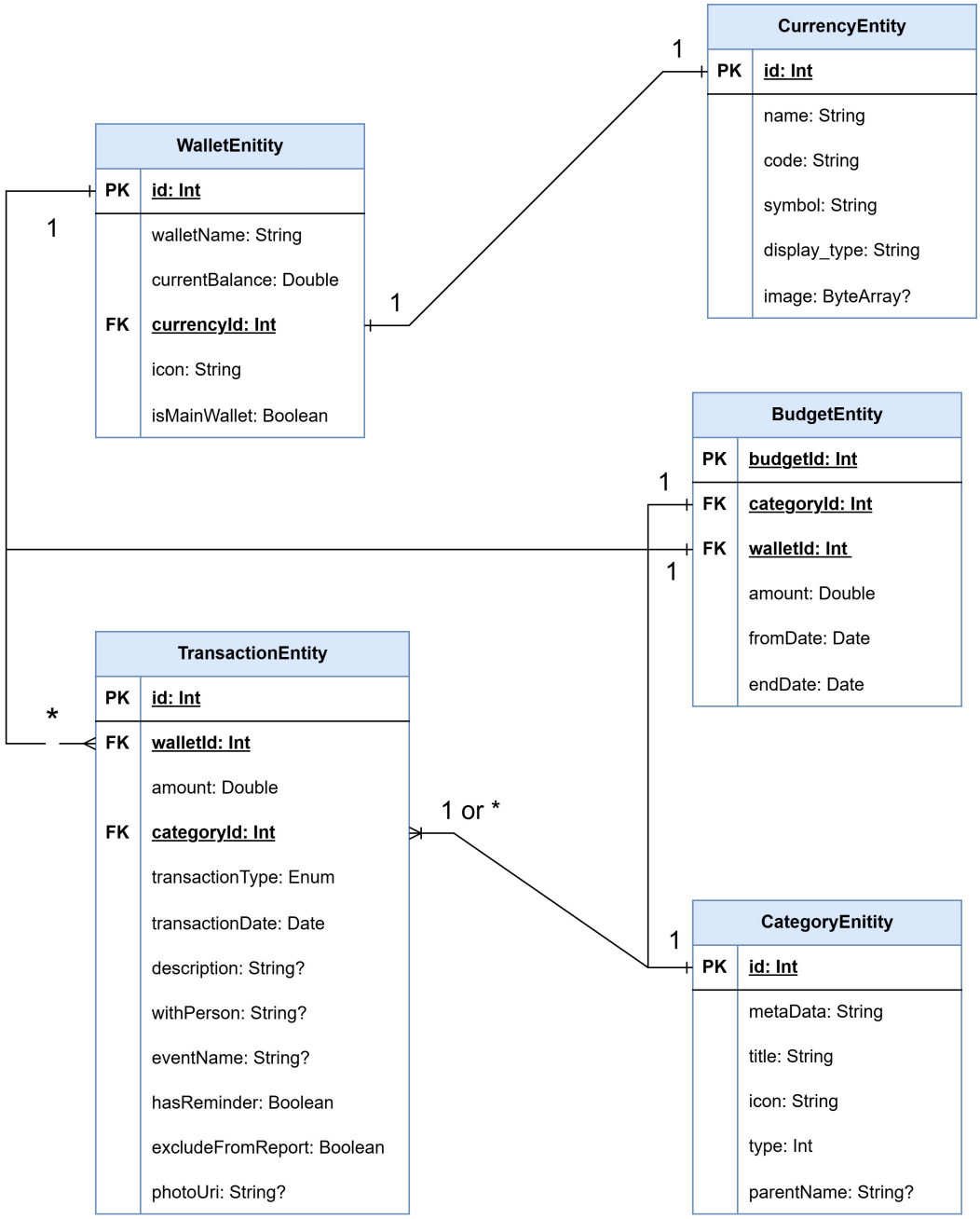

Figure 4.7: Database design diagram

4.3 Application Building

4.3.1 Libraries and Tools

Tool/Library	Version	Description
Android Studio	2023.1.1 Path 2	Integrated Development Environment (IDE) for Android app development.
Kotlin	1.5.0	Programming language used for Android development.

Continued on next page

Continued from previous page

Tool/Library	Version	Description
App Actions Test Library	N/A	Provides capabilities to enable developers to test App Action fulfillment programmatically, automating testing that would normally be done using actual voice queries or the App Actions test tool.

Table 4.5: Libraries and Tools Used in Development

4.3.2 Achievement

Packaged Products

The main packaged product of my project is the ExpenseTracker, which includes the following components:

- APK File: The compiled Android application package ready for deployment.
- Source Code: The complete source code of the project.
- Documentation: Detailed documentation including user manual and developer guide.
- Resources: All necessary resources like images, font files, and configuration files, color files, string files.

Project Statistics

Metric	Value
Total Lines of Code	18883 lines
Number of Classes	140
Number of Packages	43
Total Source Code Size	1,04 GB
Size of APK File	50 MB
Size of Resources	30 MB

Table 4.6: Project Statistics

4.3.3 Illustration of main functions

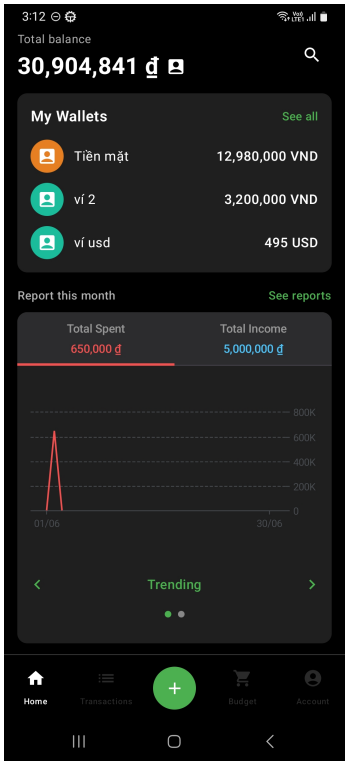

Figure 4.8: Demo Overview Screen

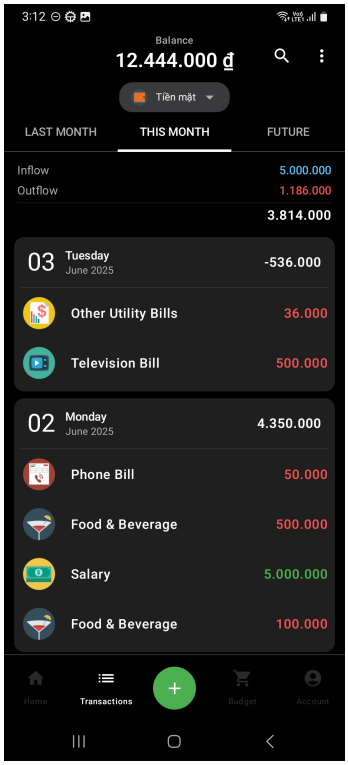

Figure 4.9: Demo List Transactions Screen

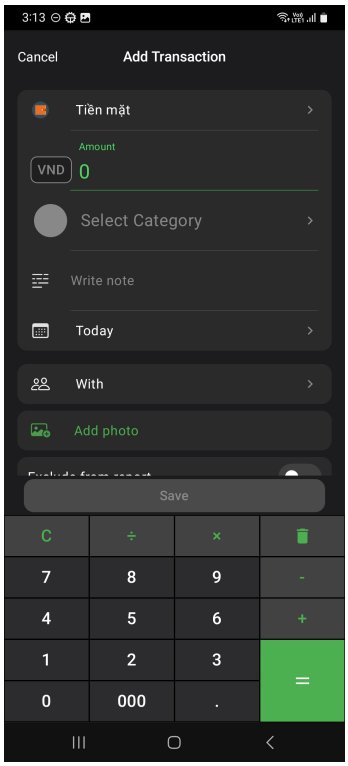

Figure 4.10: Demo Add Transaction Screen

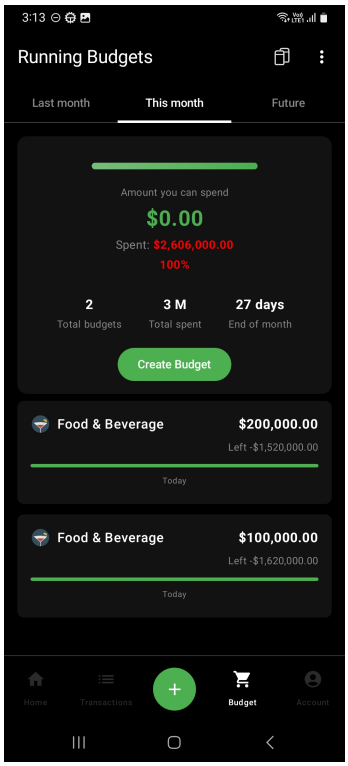

Figure 4.11: Demo Budgets Screen

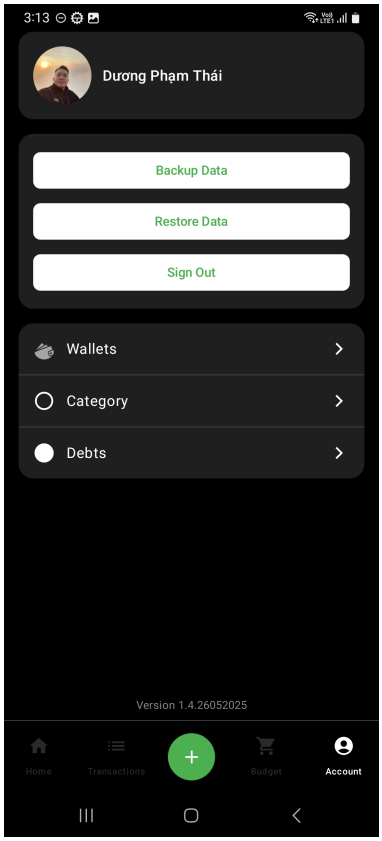

Figure 4.12: Demo Account Screen

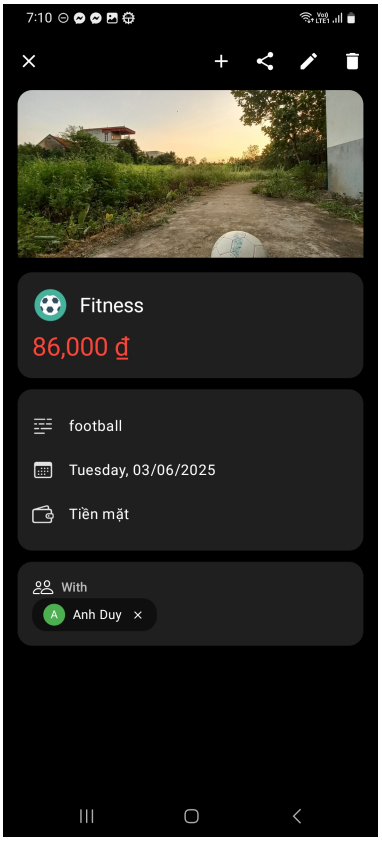

Figure 4.13: Demo Detail A Transaction Screen

4.4 Testing

4.4.1 Key Functionalities

For the expense tracker app, the critical functionalities that require validation are:

- **Add transaction by manual**
  - **Functionality Description**: Allows users to add transactions manually to ensure that the application can save the information the user enters.
  - **Validation Approach**: Check through manual form and display the correct saved transaction.
- **Add transaction by Google Assistant**
  - **Functionality Description**: Allows users to add transactions using a virtual assistant via voice so that the application can automatically identify transaction information.
  - **Validation Approach**: Check via voice or App Actions test library and display the correct saved transaction.

- **Viewing list transactions**
  - **Functionality Description**: Allows users to view a list of transactions, users can change wallets to view transactions.
  - **Validation Approach**: Test through manual testing to ensure that the list of transactions is displayed correctly according to the design and change wallets to view the list of successful transactions.

4.4.2 Summary of Validation Results

After using and validating the functionalities, the results are summarized as follows:

- **Validation Results:**
  - In practical usage scenarios, every functionality operated as anticipated.
- **Analysis of Issues, if any:**
  - Informal testing did not reveal any significant problems or concerns.
  - Minor interface issues were observed on lower resolution devices.

4.5 Deployment

- **Deployment Model**: The application is deployed on a Samsung and a Xiaomi devices for real-world testing:
  - Xiaomi Redmi 5 (5.1 inch)
  - Samsung A32 (6.3 inch)
- **Testing Methods**: Real-world testing involved using the application extensively on the specified devices to evaluate performance, usability, and user experience across different screen sizes and hardware configurations.
- **Deployment Test Results**:
  - **Number of Users**: Initially tested with a 2 beta testers using both devices.
  - **User Feedback**: Overall positive feedback on usability and performance. Some users reported UI responsiveness issues on Xiaomi Redmi 5 due to old hardware.

This chapter provides a detailed summary of the application's design, implementation, and evaluation phases. The Architecture Design section describes the overall structure and technology choices. Detailed Design focuses on specific parts, like class diagrams and database schema. Application Building highlights the coding and integration, stressing iterative development. Testing methods, including

unit and integration tests, are discussed in terms of ensuring functionality and reliability. Finally, Deployment covers how the application was launched on different devices, with a look at server configurations and performance testing.

5.1 Utilizing Hilt, MVI, and Clean Architecture for application design

5.1.1 Problem

In modern Android applications, organizing source code clearly, easily extensible, easy to test is a big challenge for programmers. Traditional architectures such as Model - View - Controller or Model - View - Presenter have difficulty maintaining large codebases, leading to the phenomenon of "God Class" or difficult to reuse. Newer can mention Model - View - ViewModel but this model still has problems with simple applications and can increase complexity, especially this model uses two-way data flow between View - ViewModel which will be difficult to debug with complex applications. And then MVI (Model-View-Intent) is an architectural variation inspired by unidirectional data flow, making UI state clear, consistent, easy to debug.

Clean Architecture (Robert C. Martin) proposes a model that clearly separates business layers and the integration of MVI with Clean Architecture is not popular and lacks standard documentation, leading to difficulties in finding the right implementation direction mainly based on personal experience. In addition, dependency injection in layered architectures requires powerful and synchronous tools. Android currently encourages the use of Hilt to support DI in a simple way, compatible with Jetpack and Compose. However, the combination of Hilt + MVI + Clean Architecture has no official pattern, requiring self-design of a suitable model.

The problem is: How to build an architecture that is clear, flexible, easy to upgrade, and takes advantage of Hilt and Compose? This is one of the contents that I like the most and spend a lot of time researching, testing, and adjusting.

5.1.2 Solution

According to Clean Architecture, the system will be designed into 3 main layers: Domain, Data, Presentation. The Domain layer's function will contain independent UseCases, pure logic and especially not dependent on Android, making it easy to test. The Data layer will contain repositories and data sources for use and connection to Room and database. The Presentation layer will be applied with the MVI model with main components such as: State, Intent, Event, Reducer. State represents the entire UI state connected to Intent which is user interaction. Event is like a part to display notifications or make requests on the UI such as switching screens or displaying notifications. Reducer is the ViewModel which is considered

as the place where the logic updates the state. ViewModel is separated into 2 clear components: handling intent (Intent Handler) and managing state (State Store). In addition, use Jetpack Compose to easily link State to UI and ensure UI updates automatically when state changes.

Organize the project into small modules corresponding to each layer, convenient for expansion and testing. Integrate Hilt to inject repository into UseCase, UseCase into ViewModel, ViewModel into Composable. This ensures dependencies are controlled and easy to change when needed. Design standard one-way data flow from UI to Intent to ViewModel to UseCase to State to UI.

5.1.3 Results

After applying the solutions, the application has a clear structure, making it easy to add new features without affecting other parts. The UI state is centrally and consistently managed, helping to limit bugs related to incorrect UI updates. It is easy to track the processing logic from UI to UseCase through Intents. The application performance is good, because the components are loaded in a reasonable scope thanks to Hilt (ActivityScoped, ViewModelScoped...). Learning and getting used to it was a bit difficult at first, but when deployed, it was effective and saved time later. This architecture can be reused for many different types of applications with little modification. Ready to expand to a multi-module architecture or a larger-scale application thanks to its high layering. This architecture has played the role of a "backbone" that helps me easily integrate Google Assistant and other complex features in the project.

5.2 How Google Assistant interacts with apps

5.2.1 Problem

In the era of increasingly popular AI and virtual assistants, integrating apps with Google Assistant is a trend to enhance user experience, changing the perspective of long-time phone users. Traditional spending management apps require users to perform manual operations such as opening the app, manually entering spending data, which is inconvenient and easy to forget for those who are not familiar with spending control. The problem is: how can users add spending, perform functions or search for information in the app quickly with just their voice?

Google introduced App Actions as a way to integrate Android apps with virtual assistants, allowing users to give voice commands to call functions in the app. Integrating Google Assistant is not as simple as adding UI, but it requires mapping logic through App Actions and Built-In Intents (BII). App Actions require explicit declaration of capabilities in the **shortcuts.xml** file and use of **Built-In-**

Intents defined by Google (e.g. actions.intent.OPEN _FEATURE) or user-defined ones approved by Google. This integration requires a deep understanding of Deep Links, Android Manifest, and intent filters, which is not common for developers who have never implemented the Assistant. Google's official documentation is quite fragmented and often lacks comprehensive examples for non-commercial use cases like "personal expense management". Additionally, it is necessary to ensure that the app handles the intent correctly, even when the app is in the background or not running. Having Google Assistant be able to correctly determine the parameters required for an inent (slot-filling) is important to avoid misunderstandings by the Assistant and many new developers have difficulty understanding the internal workings of Google Assistant. With this problem, the goal is to help users "speak to do" instead of manually operating on the app.

This is a modern, smart interaction direction, bringing high practical value and suitable for modern application trends, and I spent a lot of time learning, testing and adjusting. Although after submitting the application to Google for approval for actual implementation, the application was not successful due to objective reasons, the successful testing helped improve the application and was the highlight of the graduation project.

5.2.2 Solution

To solve this problem, let's first understand App Actions. Assistant App Actions allow users to launch and control Android apps with their voice. App Actions enable deeper voice control, allowing users to launch your app and perform actions like: launching features from Assistant, displaying app information on a Google Surface, suggesting voice shortcuts from Assistant.

App Actions use built-in intents (BIIs) to trigger these Intents and other use cases across common action categories. These built-in intents (BIIs) will allow your app to demonstrate to Google that it has the ability to handle the appropriate user request. By declaring capabilities in the "shortcuts.xml" file and mapping intent parameters to execution methods, I enabled Google Assistant to launch the app in a specific screen, in response to a query, and assist the user in completing a task.

The way App Actions work is straightforward, but difficult to implement. App Actions enhance the functionality of the app with the Assistant, allowing users to access app features using voice. When a user invokes an App Action, Assistant matches the query to a BII declared in the shortcuts.xml resource, and then launches the app on the requested screen or displays an Android widget.

I declare BII in my app using the Android function element. When I upload an app to Google Play using the Google Play Console, Google registers the features declared in your app and users can access them through the Assistant.

For example, I might provide the ability to view a list of transactions in my app. When a user says "Hey Google, open a list of transactions in the Expense Money App" the following steps occur:

1. The Assistant performs natural language analysis of the query, matching the semantics of the request to a predefined pattern of BII. In this case, actions.intent.OPEN _FEATURE matches the query.
2. The Assistant checks to see if that BII has ever been registered for the app and uses this configuration to determine how to run it.
3. The Assistant creates an Android intent to launch the in-app destination of the request using the information I provide in <capability>. Assistant will extract the query parameters and pass them as an extra in an intent created on Android.
4. Assistant responds to the user request by launching the intent created on Android. I can configure the intent to launch an in-app screen or display a widget in Assistant.

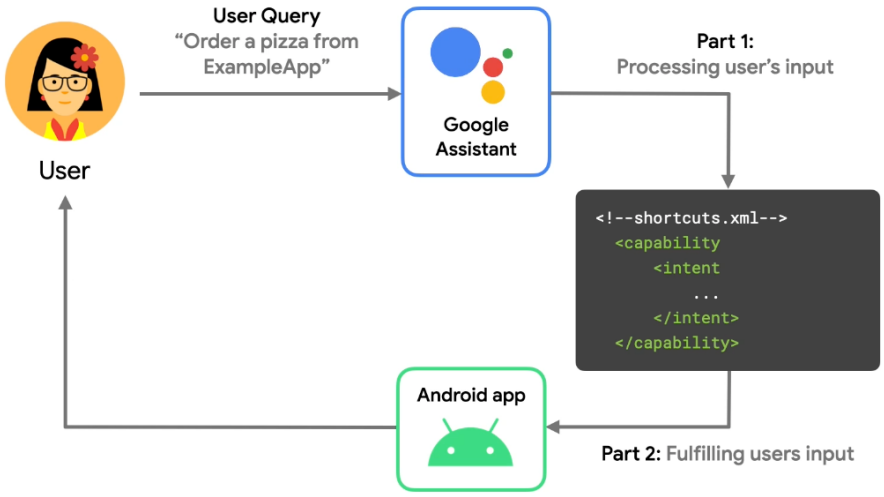

Figure 5.1: Example of a user query flow in In-app Actions.

The environment setup requirements for testing apps are also strict and require a lot of work for developers. Prepare your development environment by configuring the following:

1. Sign in to Android Studio (version 4.0 or later).
2. Sign in to the Google app on your Android test device with the same account.
3. With that account, register Play Console access to the app package you are uploading for testing.
4. Open the Google app on your Android test device and complete the Assistant setup process.
5. Enable device data sync on the test device.

Now that we have a basic understanding of how App Actions and Google Assistant work, we can rest assured that this solution will be implemented in the application with two parts: Google Assistant interaction to fill data into the application and app shares data with Assistant.

For the first part of Google Assistant filling data into the app, I need to ensure the user experience: when the app is opened, the fields need to be pre-filled so that the user just needs to confirm and press "Save". The challenge lies in: configuring the correct Intent for Google to understand, mapping the correct data (e.g. amount, category, date), the app must handle the sent intents correctly.

The main solution to configure an Intent here is:

1. In the **shortcuts.xml** file, I define a **<capability>** with **"name"** being a built-in intent or a custom intent. For example: **"custom.actions.intent.ADD_EXPENSE"** is a custom intent that allows Google Assistant to understand that your app supports the "add expense" action.

```xml
<capability
    android:name="custom.actions.intent.ADD_EXPENSE"
    app:queryPatterns="@array/expenseQueries">
    <intent
        android:action="android.intent.action.VIEW"
        android:targetClass="com.duongpt.expensetracker.features.money.ui.navigation.MainActivity"
        android:targetPackage="com.duongpt.expensetracker">
        <url-template android:value="expensetracker://addexpense{?amount,category,date}" />
        <parameter
            android:name="amount"
            android:key="amount"
            android:mimeType="https://schema.org/Number" />
        <parameter
            android:name="category"
            android:key="category"
            android:mimeType="https://schema.org/Text" />

        <parameter
            android:name="date"
            android:key="date"
            android:mimeType="https://schema.org/Date" />
    </intent>
</capability>
```

Figure 5.2: An intent configured in the file shortcuts.xml

2. Define a query pattern called **"queryPatterns"** with values like: **@array/-expenseQueries**, which contains similar sentences with the same meaning of wanting to add a transaction with amount, category, time. The queryPatterns attribute links **<capability>** to an array of natural query patterns. This helps Google Assistant understand and map the user's commands to the appropriate intent

3. Inside <capability>, I define an **<intent>** with the necessary **<parameter>** parameters. Each <parameter> has the **android:name**, **android:key** and **android:mimeType** attributes so that Google Assistant knows how to pass the correct type of data to the app. For example: amount: Amount spent, category: Spending category, date: Spending date.

4. Deep Link template (data path) is the syntax for Google Assistant to format the underlying data into a URI. For example, the sentence `"I spent 50000 on lunch today"` becomes `"expensetracker://addexpense?amount=50000&category=lunch&date=2025-06-01"`.

5. Google Assistant will pass data to the app when the user says a command that matches **"@array/expenseQueries"**. Google Assistant will then recognize the Intent (e.g. custom.actions.intent.ADD_EXPENSE), then extract the parameters from the command (e.g. amount, category, date). Next, Google Assistant sends the intent with these parameters to the app via a deep link or an

extra intent (e.g. expensetracker://addexpense?amount=50000&category=lunch &date=2024-06-01).

6. On MainActivity, the app receives data from Intent via deep link and then transfers it to Navigation, where a suitable deepLinks is set up to process the received data.

```
composable(
    route = Screen.AddTransaction.route,
    deepLinks = listOf(
        navDeepLink { this: NavDeepLinkDslBuilder
            uriPattern =
                "expensetracker://addexpense?amount={amount}&category={category}&date={date}"
        },
        navDeepLink { this: NavDeepLinkDslBuilder
            uriPattern = "expensetracker://add_transaction"
        },
    )
) { this: AnimatedContentScope  it: NavBackStackEntry
    val amount = it.arguments?.getString( key: "amount")
    val category = it.arguments?.getString( key: "category")
    val date = it.arguments?.getString( key: "date")


    AddTransactionScreen(
        onCloseClick = { navController.popBackStack() },
        navController = navController,
        initialAmount = amount,
        initialCategory = category,
        initialDate = date
    )
}
```

Figure 5.3: The app receives data from Intent via deep link

7. After getting the data, I use ViewModel to put it into AddTransactionState, displaying it on the AddTransactionScreen UI.

8. The data is displayed: amount, category, date and the user just needs to press the "Save" button to complete.

In the second part, Google Assistant queries data from the application, also known as app shares data with Assistant. Here, users not only want to enter data by voice but also want to query information from the application via Google Assistant. To meet this, the application needs to not only receive intents from Assistant, but also process internal logic, and send back the resulting data displayed directly on the Google Assistant interface. A normal Android App cannot actively send data directly to Assistant but needs to use the capabilities of App Actions combined with built-in intents and capability fulfillment.

Solution to do this:

1. Use the **<capability>** tag in **shortcuts.xml** to describe the actions that the application supports (for example, query total spending, balance, most spent category, ...).

```
84      <capability
85          android:name="custom.actions.intent.TOTAL_EXPENSE"
86          app:queryPatterns="@array/dateExpenseQueries">
87          <app-widget
88              android:identifier="TOTAL_EXPENSE_WIDGET"
89              android:targetClass="com.duongpt.expensetracker.widgets.TotalExpenseAppWidgetProvider">
90              <parameter
91                  android:name="date"
92                  android:key="date"
93                  android:mimeType="https://schema.org/Date" />
94
95          </app-widget>
96          <intent
97              android:identifier="TOTAL_EXPENSE_FALLBACK"
98              android:action="android.intent.action.VIEW"
99              android:targetClass="com.duongpt.expensetracker.features.money.ui.navigation.MainActivity">
100         </intent>
101     </capability>
102
```

Figure 5.4: An intent configured in the file shortcuts.xml

2. Define **custom intents** in the **shortcuts.xml** file so that Google Assistant can recognize specific queries of the application.
   For example: `"custom.actions.intent.TOTAL_EXPENSE"`
3. Define a query pattern called **"queryPatterns"** with values consisting of similar sentences with the same meaning of wanting to see some data in the app.
   For example: `"What is the total spending today?"`
4. Use **<app-widget>** in shortcuts.xml to associate the intent with the **AppWidgetProvider**, allowing Assistant to get data directly from the widget. When the user asks Google Assistant a query (for example: "What is the total spending today?"), Assistant will search for the declared intents, and if the intent is associated with <app-widget>, Assistant will get the data directly from the widget without opening the app.
5. Define **<intent>** with the **android:identifier** attribute as a fallback, allowing Assistant to open the app and pass data when it cannot get it from the widget.
6. Create corresponding **AppWidgetProvider** classes for each query type (e.g. TotalExpenseAppWidgetProvider, BalanceAppWidgetProvider, etc.). In AppWidgetProvider, implement the logic to retrieve data from the database or repository and update the widget interface.
7. Register the widget in **AndroidManifest** and ensure the necessary access

rights.

```
53      <receiver
54          android:name="com.duongpt.expensetracker.widgets.TotalExpenseAppWidgetProvider"
55          android:exported="true">
56          <intent-filter>
57              <action android:name="android.appwidget.action.APPWIDGET_UPDATE" />
58          </intent-filter>
59          <meta-data
60              android:name="android.appwidget.provider"
61              android:resource="@xml/total_expense_appwidget_info" />
62      </receiver>
```

Figure 5.5: Register the widget in Android Manifest

In addition, the app also supports other types of Intent as follows:

- **custom.actions.intent.ADD_EXPENSE**: Allows users to add new spending transactions via Google Assistant.
- **actions.intent.OPEN_APP_FEATURE**: Opens specific features in the app.
- **custom.actions.intent.TOTAL_EXPENSE**: Query total spending in a period.
- **custom.actions.intent.GET_BALANCE**: Query the current balance.
- **custom.actions.intent.WEEKLY_EXPENSE**: Query spending in the week.
- **custom.actions.intent.MONTHLY_EXPENSE**: Query spending in the month.
- **custom.actions.intent.CATEGORY_EXPENSE**: Query spending by category.
- **custom.actions.intent.TODAYS_INCOME**: Query daily income.
- **custom.actions.intent.TOP_SPENDING_CATEGORY**: Query the most spent category.
- **custom.actions.intent.BUDGET_USAGE**: Query budget usage status.

To test the functionality of the Intent in both of these sections, I used the App Actions Testing Tool to enter the values of the parameters, and then when the user clicks "Run App Action" in the App Actions Test Tool, this tool will create a deeplink similar to the user saying a request to the virtual assistant, and after the virtual assistant processes the user's request, it will send a deeplinks so that the application registered for deeplinks can recognize and process it. I will leave an illustration for the results of the process of Google Assistant filling data into the app and the process of Google Assistant querying data from the application in the Results section below.

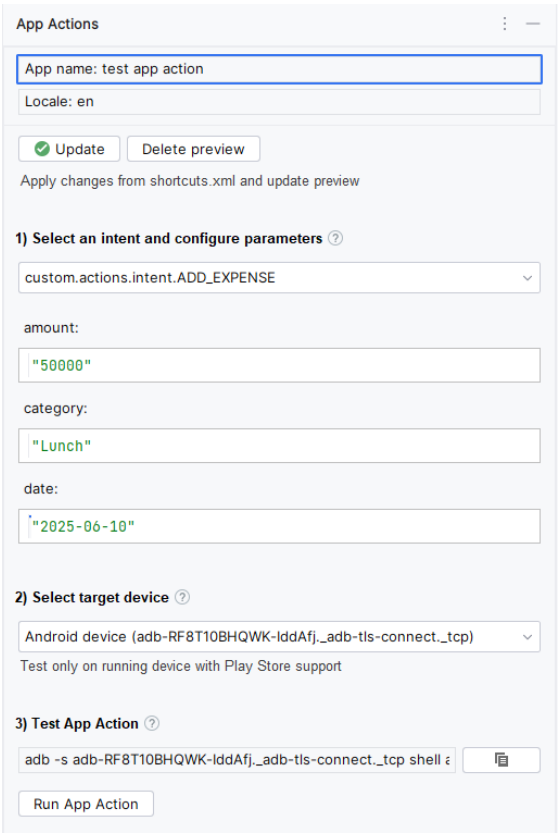

Figure 5.6: Test Google Assistant Filling Data into App

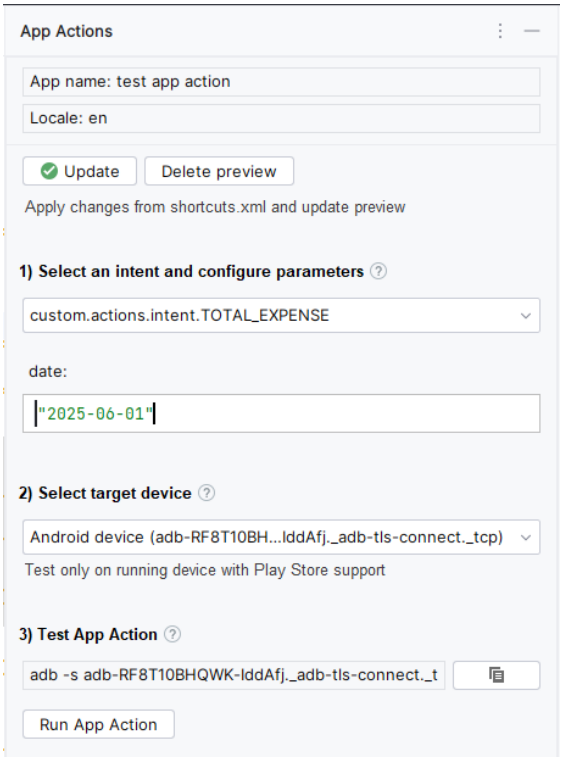

Figure 5.7: Test Google Assistant Querying Data from App

5.2.3 Results

After having a solution and successfully deploying it, users can interact with the application through voice commands to Google Assistant. The data will be automatically filled into the input form, helping to reduce time and reduce errors. Testing confirms that all input cases work stably. Below is the result of the process of Google Assistant filling data into the application

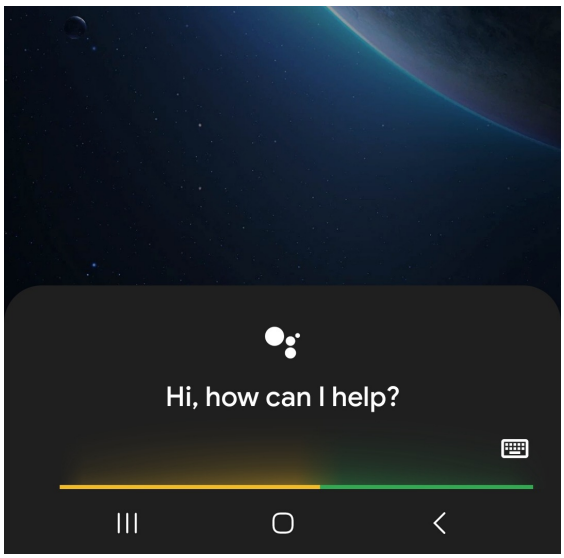

Figure 5.8: The process of Google Assistant processing requests

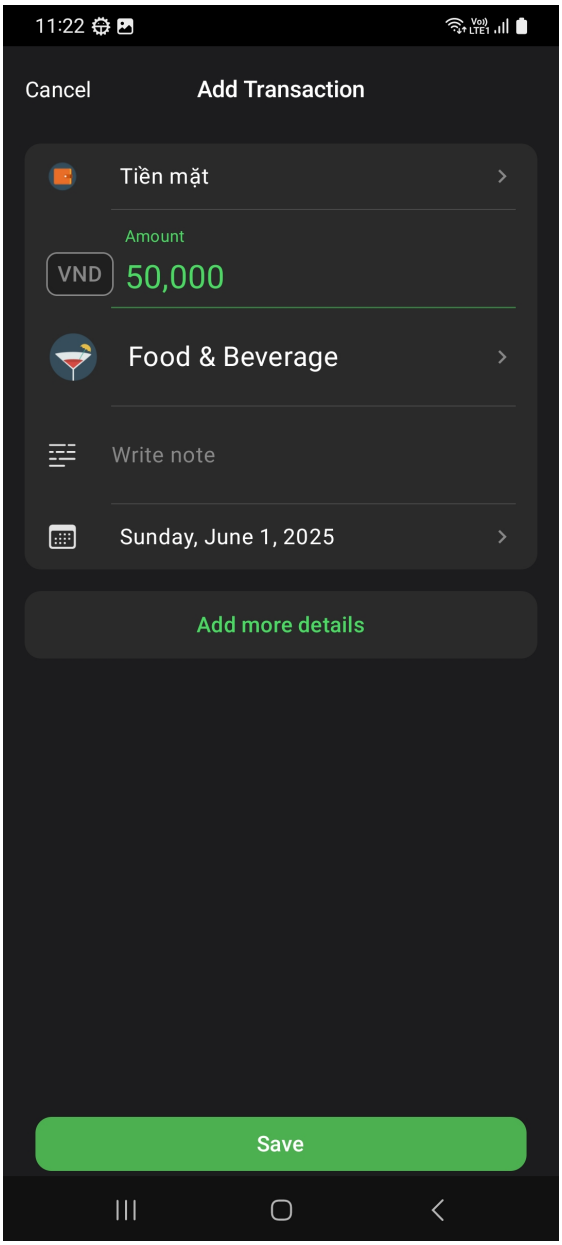

Figure 5.9: Google Assistant fills data into the fields

In addition, Google Assistant's query of data from the application is also well-responsive when the user requests any data from the application. The response process is completely automatic, without opening the application. The response results are accurate according to predefined criteria. This is a step to confirm that

the application has the ability to "reversely communicate" with Assistant, not only receiving commands but also proactively responding. Below is the image of the result of the process of Assistant getting data from the application to quickly respond to the user.

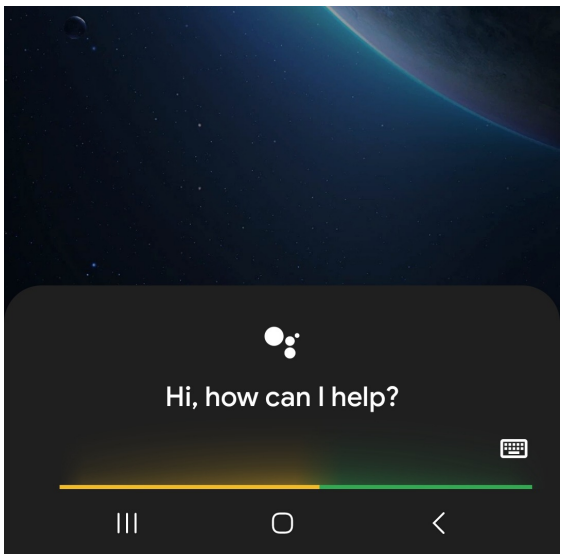

Figure 5.10: The process of Google Assistant processing requests

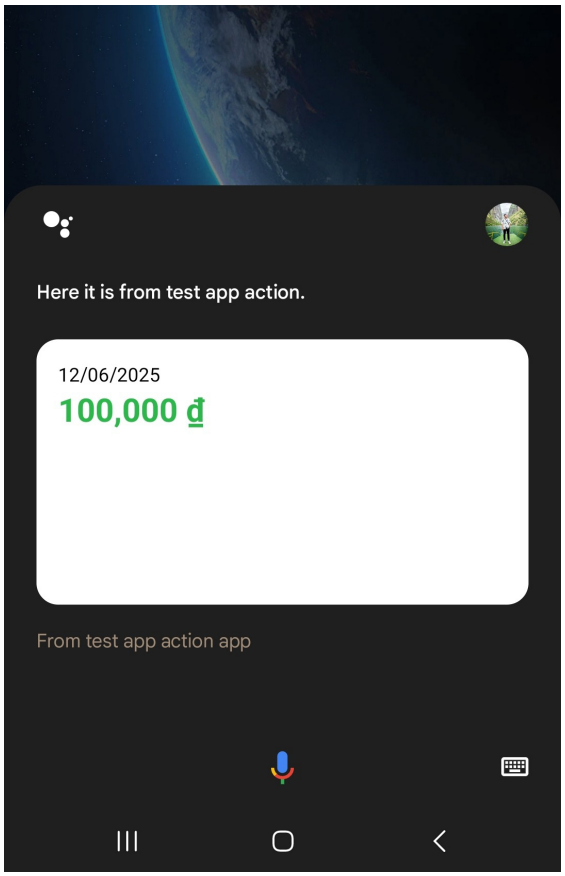

Figure 5.11: Google Assistant displays data to the user

The application allows users to perform tasks by voice and grasp the necessary information without having to go directly to the application. That contributes to significantly improving the user experience, convenient when driving, busy or not wanting to open the application. In addition, it also increases the modernity and professionalism of the application, proving compatibility with the Android ecosystem. Integrating the virtual assistant into the application does not affect the application architecture, because it is separated through deep links and intent handling. The assistant responds accurately and naturally thanks to the definition of response actions within the capabilities. It can be extended to other types of App Actions like reminders, statistics, card balance inquiries... Making the app stand out from the usual expense management apps. I learned a lot of new knowledge about Assistant, basic NLP and natural language user interaction.

5.3 Synchronize spending data across multiple devices

5.3.1 Problem

In personal spending management applications, users often want to be able to access data on many different devices (phones, tablets, etc.). However, the default Android application uses SQLite/Room for local storage, which does not have the ability to synchronize. In addition, switching devices, reinstalling the app, or losing the device causes users to lose all data if there is no backup solution. Some popular solutions such as using Firebase Realtime Database or Firestore can be redundant, complicated, and costly for simple applications. I set out to find an easy-to-deploy, lightweight, low-cost backup solution that fits the existing structure of Room. The idea came to me: if the database file (Room SQLite) could be saved to Firebase Storage, it would be possible to backup/restore it like a document file. Users will be asked to sign in with their Google account, and can proactively choose when to backup or restore. The solution needs to ensure: security, ease of use, no crashes or data conflicts. This is a practical function, highly valuable to users, and helps the application to be "cloud sync" without being too complicated.

5.3.2 Solution

To solve the synchronization problem in a simple way, I ask users to log in with their Google Sign-In account because they need a Google account to download the application, so they will also have an account to log in easily.

After logging in to the application, the "backup & restore" feature will be active. I use Firebase Authentication to authenticate, thereby creating a separate namespace for each user (eg: /backups/userId/expense_tracker.db). Because each account will have a different userId, it ensures that the directory will not be duplicated.

When the user selects “Backup”, do the following:

1. Close the Room database to ensure that the data has been written to disk.
2. Access the expense_tracker.db file in the application directory (context.getDatabasePath(...))
3. Upload this file to Firebase Storage.

When the user selects “Restore”, do the following:

1. Download the .db file from Firebase Storage (path according to userId).
2. Overwrite the current database.
3. Re-initialize the Room database after replacing.

Special case handling: If there is no backup file then display a gentle notifica-

tion. If restoring, do not allow operations on the UI to avoid conflicts.

This method ensures security because only users with the corresponding userId can download their database file. With a simple interface, consisting of only two buttons: "Backup Now" and "Restore from Cloud", anyone can use it.

5.3.3 Results

Users can easily backup or restore all data with just one click. Data synchronization between two Android devices with the same Google account has been successfully tested. In case of uninstalling the app or reinstalling the device, users can restore data immediately without loss. The average backup file is only a few hundred KB, which is suitable for storing on Firebase Storage. During the backup and download process, data is transmitted quickly. Users' privacy is protected because only they have access to their backup files. The application will be easily expanded in the future: add encryption features, automatic backup, real-time synchronization if there is a backend. The interface is designed simply, anyone can use it without instructions. The solution works stably during real-world testing with both emulators and real devices. This is a big plus for the app because it feels professional and friendly like “cloud-native” apps.

CHAPTER 6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

Compare my research or product with similar research or products. The Android expense management application that I developed is a platform that allows users to manage expenses with basic functions such as wallet management, transaction management, budget and debt management, and also supports backup and restore in case users want to save and retrieve data. To stand out in the financial management application market, my application has an additional function of direct interaction with a virtual assistant that no other application on the market has.

To evaluate my application, I researched similar and popular expense management applications that are currently available with over 1 million downloads to have an objective view. The applications I researched such as: MISA Money-Keeper, Money Tracker, Fast Budget, and Money Lover all have their own advantages and disadvantages, this comparison helps me understand the position of the product in the current market.

MISA Money-Keeper is one of the popular applications with the platform of a financial company like MISA, so the application stands out with many features for users to choose from. My application can be compared to MISA Money-Keeper in terms of not requiring paid features, a more modern interface and support for interaction with virtual assistants. However, MISA Money-Keeper stands out in terms of the number of features and simplicity. Features such as calculating personal income tax, calculating loan interest and exporting data to excel files provide more options for users in special cases. However, my application is still in its early stages, so I may deploy these features in the near future.

Money Tracker is a simple, uncomplicated application and is also one of the top download applications. The only strong point of this application is that it does not require a login to use. In addition, in terms of functionality, there is nothing more outstanding than my application. The basic functions related to wallet, transaction, report, my application also has full. The interface of this application is difficult to use, it takes a lot of time to find the main functions. The main color of the application is yellow, which is not suitable for the financial industry, which is mainly green and red. This application requires payment for some features, while my application is completely free. In particular, this application does not support interaction with virtual assistants, which is also not convenient compared to my application.

Fast Budget is an application with many downloads on Google Play, but this is an application with full functionality, supporting offline use without logging in. My application can be compared to this application in terms of interface and features. In terms of interface, this application is too simple, not clearly showing the functional components, while my application has a modern interface design, clearly dividing the interface components. In terms of functionality, this application still requires selecting the type of transaction, while my application has automatic support, users only need to select the type of category. This application is superior to my application in additional functions such as scheduling transactions, exporting report files. However, in return, my application supports working with virtual assistants.

Money Lover is also a popular application and has the most downloads among the applications mentioned. This is a powerful and almost perfect application because it has a modern interface, reasonable colors, features are arranged in a suitable visual manner, supports reporting and user reminders. My application can be compared to Money Lover in terms of the function of interacting with virtual assistants and is completely free. However, Money Lover has a big advantage in terms of stability and has gone through many updated versions, which my application has not had despite my best efforts. Money Lover is outstanding in terms of interface thanks to the in-depth analysis of the development team. Their interface is also the main idea for me to design the interface for my application thanks to its intuitive design, putting the user at the center.

Analysis of the Implementation Process

During the implementation of my thesis project, I completed the main tasks such as developing an application that supports interaction with a virtual assistant, completing the necessary functions of a spending management application, supporting backup and restore, and a simple, user-friendly interface that ensures ease of use. These tasks contribute to ensuring that users have a good experience on all devices. In addition, I also learned how to use the Google Play Console, which is where I submit my application to the app market and go through many rounds of Google app testing before the application is uploaded.

However, there are things I have not implemented, such as the feature of adding voice transactions in the application, which limits the options for users who want to add transactions. The application also does not support advanced features such as spending suggestions using artificial intelligence, exporting report files, and supporting multiple login methods. These limitations may reduce the ability to com-

pete with high-end expense management apps.

My most important contribution in this project was to develop an app with a friendly interface, supporting virtual assistant interaction features. The app helped me master and understand the skills of Android app development.

Careful planning and effective task distribution were crucial for the timely completion of the project. By clearly setting objectives and timelines, tasks were executed efficiently. Ongoing testing and performance optimization of the app guaranteed a superior user experience. Key actions included code optimization and routine bug checks. Regularly update and draw insights from similar apps to enhance your product. The mobile application market industry is ever-evolving, and staying abreast of trends is vital for competitiveness.

6.2 Future work

To complete the product, I will focus on necessary tasks such as integrating more login methods, supporting more voice transactions in the application. I will continue to improve the existing functions to make the application work more smoothly.

In addition, I will analyze new directions such as integrating artificial intelligence to support users in adding transactions, navigating functions and tracking reminders of income and expenditure activities. Use data analysis tools to better understand users' application usage habits.

During the process of implementing my graduation project, I acquired several important skills in Android app development. Although the product's current limitations, I am confident that, with a defined development strategy and insights gained, I can refine and enhance the application to better serve user needs.

The creation of a spending management application necessitates not only technical expertise but also a comprehensive understanding of user behaviors and requirements. By examining analogous products and meticulously evaluating my own strengths and weaknesses, I have identified opportunities for enhancement and advancement of the product. In moving forward, a key focus will be on refining current features and innovating new ones. Concurrently, broadening the market reach and attracting a larger user base remains a significant objective. With strategic planning and consistent effort, I am confident that my application will evolve into a more refined solution, thereby providing increased value to its users.

REFERENCE

[1] Demandsage, *Android Statistics*. [Online]. Available: https://www.demandsage.com/android-statistics/.

[2] Google, *Android Documentation*. [Online]. Available: https://developer.android.com/docs.

[3] Firebase, *Firebase Documentation*. [Online]. Available: https://firebase.google.com/docs.

[4] Google, *Material Design Guidelines*. [Online]. Available: https://material.io/design.

[5] Google, *ViewModel Overview*. [Online]. Available: https://developer.android.com/topic/libraries/architecture/viewmodel.

[6] Hilt, *Hilt Design Overview*. [Online]. Available: https://dagger.dev/hilt/