QuickPay Report 1 - Full Report

Group B

Chanrattnak Mong, Seanglong Lim, Seth Tharo Hour, Sok Sreng Chan

Fort Hays State University

2025F CSCI441 A Software Engineering

Dr. Mike Mireku Kwakye

September 21, 2025

GitHub Repository: https://github.com/CSCI441-QuickPay/QuickPay-MobileApp

Table of Contents

Individual Contributions Breakdown

Sections	Chanrattnak Mong	Seanglong Lim	Seth Tharo Hour	Sok Sreng Chan
Cover Page	x	x	x	x
Individual Contributions Breakdown			x	
Table of Contents			x	
Work Assignment	x		x	x
1a. Problem Statement	x			x
1b. Decomposition into Sub-problems				x
1c. Glossary of Terms		x		
2a. Business Goals		x		
2b. Enumerated Functional Requirements	x		x	
2c. Enumerated Nonfunctional Requirements	x		x	
2d. User Interface Requirements	x			
3a. Stakeholders				x
3b. Actors and Goals				x
3ci.Casual Description		x	x	
3cii. Use Case Diagram		x		
3ciii. Traceability Matrix (1) - REQ to UC			x	
3civ. Fully Dressed Description			x	
3d. System Sequence Diagrams	x			
4a. Preliminary Design	x			
4b. User Effort Estimation	x			
5a. Identifying Subsystems	x	x		
5b. Architecture Styles	x	x		
5c. Mapping Subsystems to Hardware		x		
5d. Connectors and Network Protocols		x		x
5e. Global Control Flow		x		
5f. Hardware Requirements		x	x	x
6. Project Management/Plan of Work	x	x	x	x

Sections	Chanrattnak Mong	Seanglong Lim	Seth Tharo Hour	Sok Sreng Chan
7. References		x		

Work Assignment

Chanrattnak Mong (Team Leader): I am certified in full-stack development, with expertise in UI/UX design and front-end development to deliver seamless and accessible user experience. Additionally, I manage data security using Firebase Authentication and Firestore and oversee CI/CD pipelines with GitLab to streamline backend development and deployment cycles. As team leader, I oversee both iOS and Android development to ensure cross-platform consistency and performance, while focusing my own development efforts on iOS. I coordinate communication, guide technical and design decisions, and ensure tasks are distributed fairly and completed on time to achieve smooth, high-quality delivery.

Seanglong Lim: My strengths are in backend development with Node.js and Express.js, along with database management using Firebase Firestore and SQL. I focus on designing secure and efficient database structures that support reliable system performance. By integrating Firestore with backend APIs, I help ensure seamless data flow between the server and the client side. I also troubleshoot issues related to performance and security, contributing to the overall scalability and dependability of our applications.

Seth Tharo Hour: I specialize in quality assurance, security testing, and deployment. My responsibilities include conducting rigorous testing to find bugs and vulnerabilities, then applying solutions to improve system reliability. I also implement security measures that help protect user data and maintain compliance with best practices. Using CI/CD pipelines, I streamline deployment, so updates and new features are rolled out smoothly without disruption. My work ensures the team's projects are both stable and secure before reaching users.

Sok Sreng Chan: I work primarily with React Native, the Expo Framework, and TypeScript to build responsive, cross-platform applications. My role also includes mobile optimization, ensuring the applications are intuitive and accessible on both iOS and Android devices. I contribute by developing reusable components, refining layouts, and enhancing mobile responsiveness. I also focus on performance improvements, so the user experience feels smooth and polished. By combining design and functionality, I help create mobile apps that are both engaging and reliable.

Customer Problem Statement of Requirements

I. Problem Statement

A. Fragmented Finances in Daily Life

Managing personal finances today can feel overwhelming. Most of the tools we have encountered either do not connect directly to our accounts or require manual entry for every transaction. While we seek to maintain control over our finances, we often find ourselves constantly chasing numbers.

The budgeting applications we have used fail to provide a clear view of how our funds are allocated between needs, wants, and savings. Although they offer categories, these are not always intuitive, and we cannot immediately visualize the balance of our money. Moreover, because transactions occur daily, the balances of each budget category are constantly changing, and current tools do not allow us to track these changes in real time. Consequently, we often rely on spreadsheets, spending hours entering expenses, updating balances, and reallocating funds across categories. However, life does not pause for manual updates; if we miss updating for a day or two, the figures no longer reflect our actual account balances. This discrepancy forces us to make financial decisions without full visibility, sometimes resulting in unintentional overspending.

We are not facing these challenges alone. While individual experiences may vary, a shared frustration persists: the existing tools are not designed to align with the way we live.

1. Emily

Emily is a thirty-four-year-old marketing coordinator living in the suburbs of a mid-sized city. She is married and has two young children, ages five and eight. Between balancing her full-time job, getting her kids ready for school, and making sure the household runs smoothly, Emily feels like she is always on the move. On the outside, her life looks stable, with a nice home in a safe neighborhood and enough money to cover her family's needs, but beneath the surface, managing her finances has become one of her greatest stressors.

Emily and her husband keep three separate bank accounts: one joint account for rent, mortgage, and utilities; a personal account for her husband's expenses; and a personal account for her own spending. In addition, they share a savings account that they try to contribute to each month for

emergencies and their children's future education. While the setup was originally intended to keep things organized, it has turned into a complicated juggling act. Emily constantly finds herself unsure which account to pull from when making purchases, and she worries about over drafting one account while another sits untouched.

Her morning routine often sets the tone for her financial stress. Before heading to work, she quickly glances at her bank apps to check balances. One account shows her deposited paycheck, but another is dangerously low because the utility bill was automatically drafted the night before. She immediately transfers money over, but she knows it will take a couple of days to show up. Until then, she carries a lingering sense of unease, wondering if her debit card might be declined for something as simple as groceries or gas.

a. The Strain of Manual Budgeting.

Emily has tried to maintain a family budget through spreadsheets. She tracks categories like groceries, kids' extracurricular activities, and savings goals, but every transaction requires her to manually input numbers. She sets aside thirty minutes every Sunday evening to "update the budget," but with two energetic kids demanding her attention, this process rarely goes smoothly. Sometimes she forgets to log purchases for days at a time, and when she finally sits down to catch up, the numbers do not line up with what her bank statements show.

One month, she discovered that they had overspent on takeout by nearly $300 without realizing it. Because the spreadsheet was not updated in real time, Emily thought they were still within their limits, but their bank account told a different story. That mistake forced them to dip into savings just to cover rent. It left her frustrated and ashamed, as though she had failed at something she was trying so hard to manage.

She has tried a few budgeting apps, but most of them only connect partially to her accounts or fail to categorize expenses in ways that make sense to her. They also bombard her with confusing charts or ads for credit cards, which only adds to her frustration. What Emily truly wants is a simple, intuitive way to see where her money is going, without spending hours updating spreadsheets or second-guessing whether the information is current.

b. Group Payments and Everyday Stress.

On top of her own household expenses, Emily frequently faces the headache of splitting costs with others. For example, she and a group of other parents often organize after-school activities for their children. Sometimes one parent covers the cost of snacks, tickets, or supplies, and then everyone else is supposed to pay them back. But the process rarely goes smoothly.

Last month, Emily covered the cost of admission for her daughter's entire soccer team when they went to a weekend tournament. She expected the other parents to send her their share, but it took nearly three weeks of reminders before everyone paid. In the meantime, her checking account was short by nearly $400, forcing her to put off a scheduled payment to their credit card. She hated sending reminder texts, worrying that she sounded pushy or rude, but she also could not afford to absorb the cost herself.

This is not a one-time issue; it happens almost every time she participates in group activities. Different parents prefer different payment apps, some people pay immediately while others delay, and sometimes payments get lost altogether. Each time Emily is left keeping track of who owns what, writing down names and amounts in yet another spreadsheet. What should be simple ends up feeling like unpaid part-time bookkeeping work.

c. Transparency and Trust.

Another thing that frustrates Emily is the lack of transparency in the tools she uses. One budgeting app she tried would link to her accounts, but when she made a purchase, it did not tell her which account the money came from until after the transaction was complete. More than once, she assumed it would pull from her spending account, only to discover later that it had drawn from her savings. That kind of confusion makes her feel like she is not truly in control of her money.

She also struggles with the inability to split a single payment across multiple accounts. Sometimes she wants to balance her spending by using part of the money from her Bank of America account and the rest from her Commerce account, but current tools do not give her that flexibility. Instead, she is forced to move money around manually before making a purchase, wasting time and creating even more opportunities for mistakes. If she forgets to transfer funds, she risks over drafting one account while leaving the other untouched, which only adds to her frustration.

She has also tried apps that require her to preload money into a wallet before making payments. At first, she thought it might be helpful, but the longer she used it, the more uncomfortable she felt. She worried about security risks and what would happen if the company froze her account or was hacked. The idea of handing over her family's hard-earned money to a third party made her uneasy, and she stopped using the service after just a few weeks.

Emily needs a financial system that respects her trust. She does not want to jump through hoops or wonder where her money is stored. She wants her funds to remain safe in her own accounts, with tools that act as a guide rather than a middleman.

d. The Emotional Burden.

For Emily, the biggest toll of all, this is not just the numbers; it is the stress. Money affects every decision she makes; from what groceries she buys to whether she signs her kids up for a summer camp. The lack of clarity around her finances weighs on her mind constantly.

Some nights, after the kids are in bed, she lies awake worrying about whether they are saving enough for emergencies, whether her accounts are balanced, or whether another surprise expense will throw everything into chaos. This constant state of worry bleeds into other parts of her life. She finds herself distracted at work, short-tempered with her kids, and anxious when talking with her husband about money. Instead of feeling empowered, she feels trapped in an endless cycle of managing and re-managing the same problems.

II. Decomposition into Sub-problem

Individual Customers Like Emily:

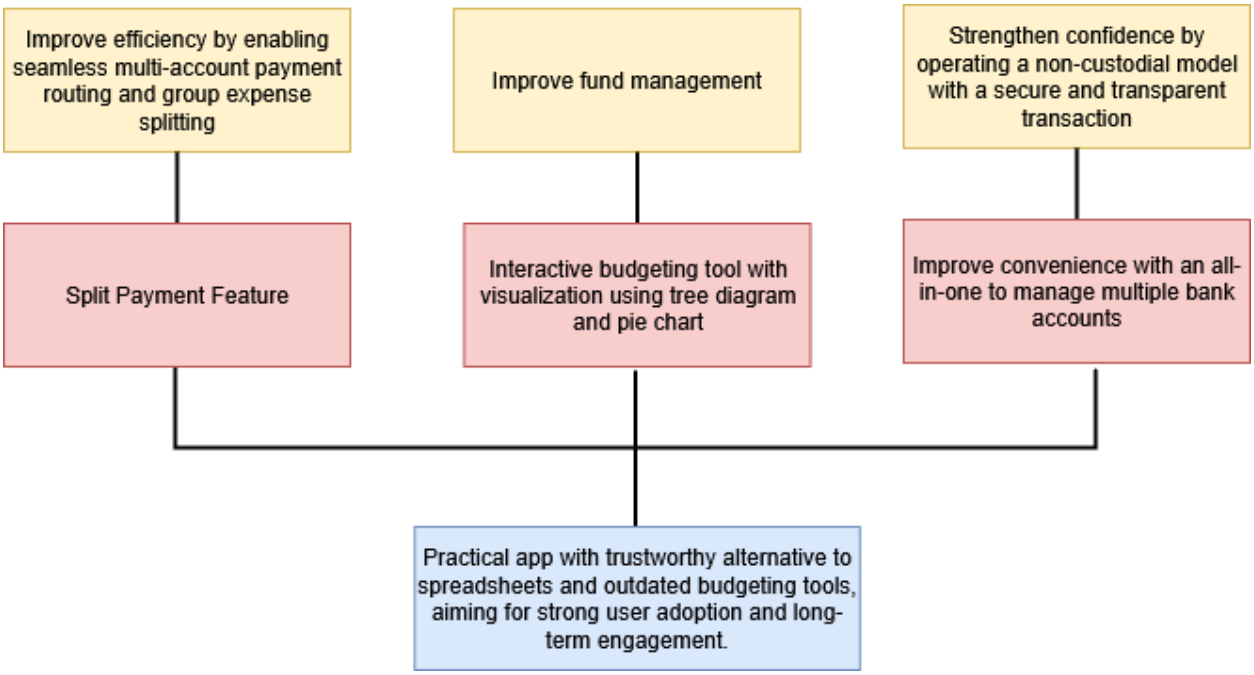
1. Customers want to see all their bank accounts in one place instead of switching between multiple apps.
2. Customers want real-time tracking of spending across categories (needs, wants, savings) to avoid overspending.
3. Customers want to reduce or eliminate the time spent manually updating spreadsheets or budgeting tools.
4. Customers want automated expense categorization so they can better understand where their money is going.
5. Customers want group payments to be easier, with clear tracking of who has paid and who still owns.
6. Customers want full transparency about which account payment is drawn from to prevent confusion.
7. Customers want the flexibility to split a single payment across multiple accounts
8. Customers want to keep control of their money without being forced to move funds into custodial wallets.
9. Customers want peace of mind and reduced financial stress through accurate, real-time insights.

III. Glossary of Terms

- **Administrator:** An individual with the authority and responsibility to configure and manage a system.
- **API (Application Programming Interfaces):** The set of rules that enable applications to communicate with each other. In our system context, we use API keys to connect with banks and payment platforms.
- **Authentication:** The process of verifying a user's identity before granting access to the system.
- **Automatic Budgeting:** The app organizes your spending into categories (like groceries, rent, or entertainment) without you having to type everything in.
- **Bill/Expense Splitting:** A way to easily divide shared costs (like dinner with friends) so everyone pays their share without awkward reminders.
- **Budget Visualization:** A graphical representation that shows how funds are divided.
- **Dashboard:** The main interface where users can view balances, spending categories, and financial summaries briefly.
- **Expense Category:** A label used to organize spending, such as groceries, rent, dining out, or savings.
- **Instant Payment:** Sending or receiving money right away, without waiting for transfers to clear.
- **Non-Custodial:** A system design where the customer’s money always stays in their own bank accounts, never held by the app itself.
- **Transparency:** Always knowing exactly where your money is coming from and where it is going with no hidden steps or confusion.
- **Plaid API:** Securely links user bank accounts and retrieves balance and transaction.[1]
- **Firebase Authentication API:** Provides secure, multi-method login (email, phone, social login) and user identity management for web and mobile apps. [2]
- **Firebase Cloud Messaging (FCM):** Delivers mobile push notifications. [3]
- **Google Charts API:** Generates budget graphs and spending flow visualizations. [4]

Goals, Requirements, and Analysis - (System Requirements and Analysis)

I. Business Goals



With Quickpay first we want to Improve efficiency in finance management by enabling seamless multi-account payment routing and group expense splitting, improve fund management and strengthen confidence by operating a non-custodial model with a secure and transparent transaction. With these improvements we introduced to society we implemented features such as split payment features, an interactive budgeting tool and an all-in-one feature to manage multiple bank accounts. Our app QuickPay will be a trustworthy alternative to spreadsheets and outdated budgeting tools, aiming for strong user adoption and long-term engagement.

II. Enumerated Functional Requirements

Identifier	Priority	Requirement
REQ-1	5	The system shall provide an interface for linking multiple bank accounts and allow users to set preferences for automatic transactions on essentials such as rent, electricity etc.
REQ-2	5	The system shall support QR code–based payments that allow personal payments and efficiently support payment between multiple parties.
REQ-3	5	The system shall include a visual budgeting tool that displays spending and allocations using an intuitive tree or flow-chart.
REQ-4	5	The system shall eliminate the need for third-party fund storage by connecting directly to users' existing bank accounts, ensuring full user control over funds and providing complete transparency about which accounts are used for each transaction.
REQ-5	5	The system shall provide a digital wallet interface that allows users to view balances across all linked accounts in one place and manage transactions from a unified dashboard.
REQ-6	4	The system shall provide group expense management with automatic bill-splitting and payment reminders.
REQ-7	3	The system shall support customizable alerts for low balances, unusual activity, or upcoming bills.
REQ-8	4	The system shall let users add other users as their favorites for easy payment.
REQ-9	5	The system shall allow users to view and update information in their profile and adjust the app setting.

III. Enumerated Non-functional Requirements

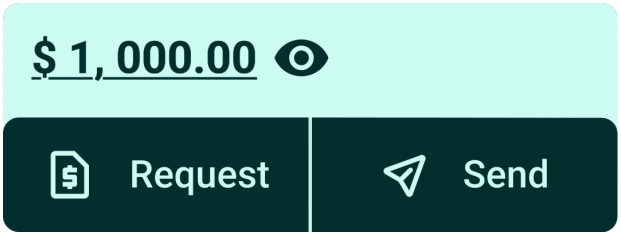
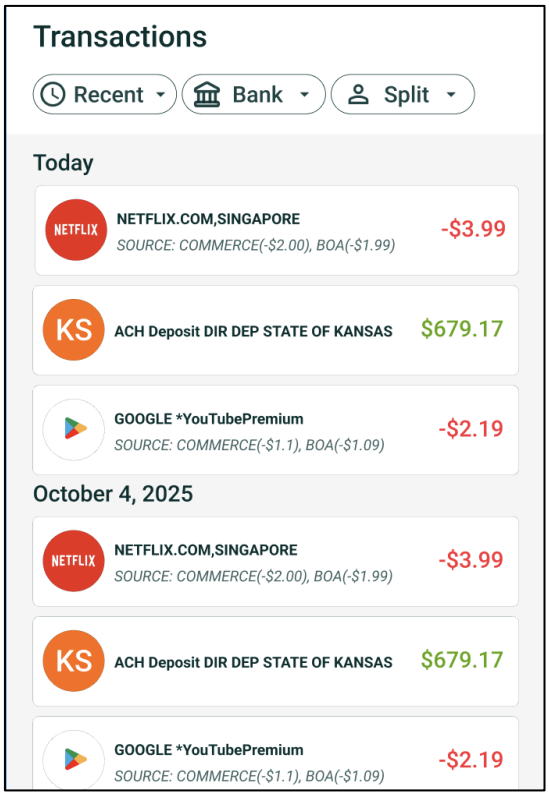
Identifier	Priority	Requirement
REQ-10	5	The system shall protect user data with encryption in transit (TLS 1.3) and at rest (AES-256), secure APIs (OAuth 2.0, OpenID Connect), and multi-factor authentication (MFA).
REQ-11	4	The system shall process payments within 2–3 seconds, generate/scan QR codes in less than 1 second, and maintain a 99.95% transaction success rate.
REQ-12	5	The system ensures 99.9% uptime with automated backup, disaster recovery (RPO < 15 min, RTO < 1 hour), and failover across multiple data centers.
REQ-13	3	The system should provide a user-friendly interface with clear typography, readable text, and simple navigation for accessibility.
REQ-14	4	The system shall collect only necessary transaction metadata, provide GDPR/CCPA-compliant consent, and allow users to download or delete their data.

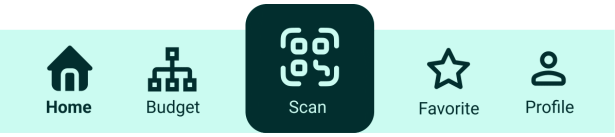
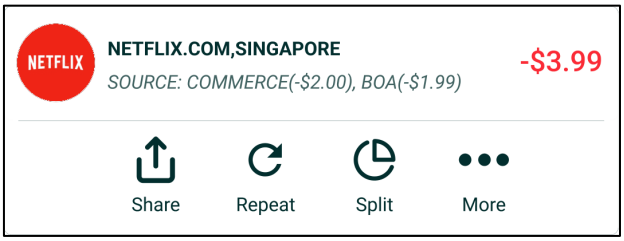
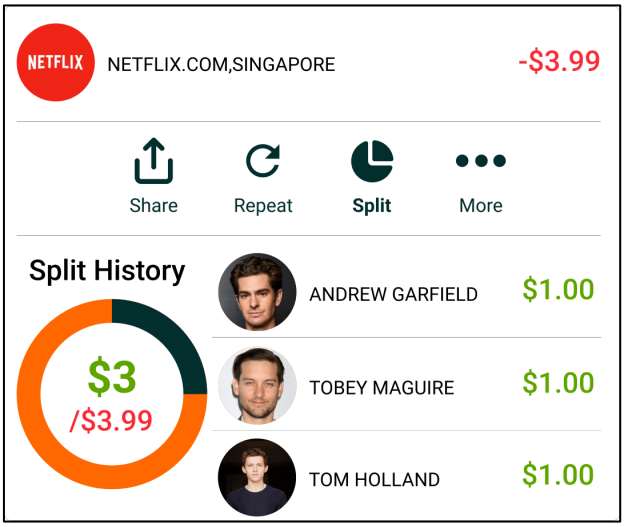
FURPS+ Table

Category	Requirement	Description
Functionality	REQ-1 to 9	Core system capabilities: linking accounts, QR code payments, budgeting visualization, net worth views, non-custodial operations, bill-splitting, and alerts.
Usability	REQ-13	Ensures a simple, intuitive, and easy-to-navigate interface for users.
Reliability	REQ-12	High system availability with minimal downtime.

Category	Requirement	Description
Performance	REQ-11	Fast execution of payments and system actions (within seconds).
Security	REQ-10	Strong authentication and protection against unauthorized access.
Privacy (+)	REQ-14	Collects minimal user data, with user-controlled permissions and transparency.
Compliance (+)	REQ-5	Non-custodial design ensures compliance with financial regulations.

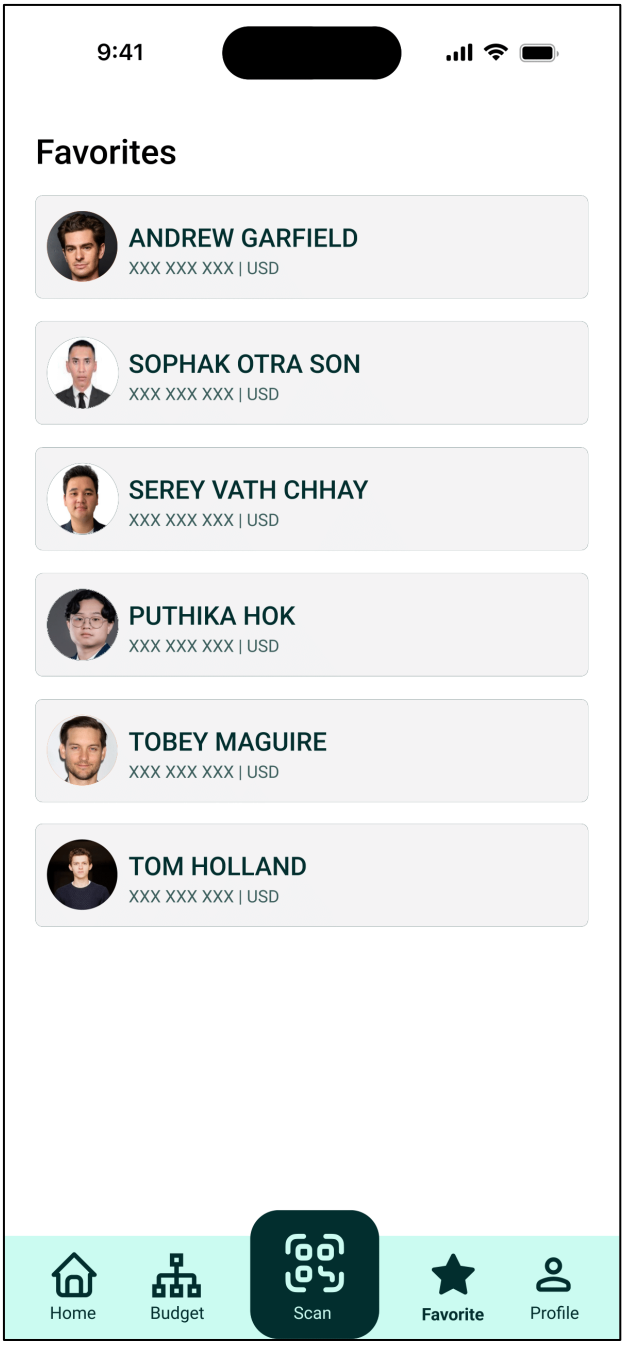
IV. User Interface Requirements

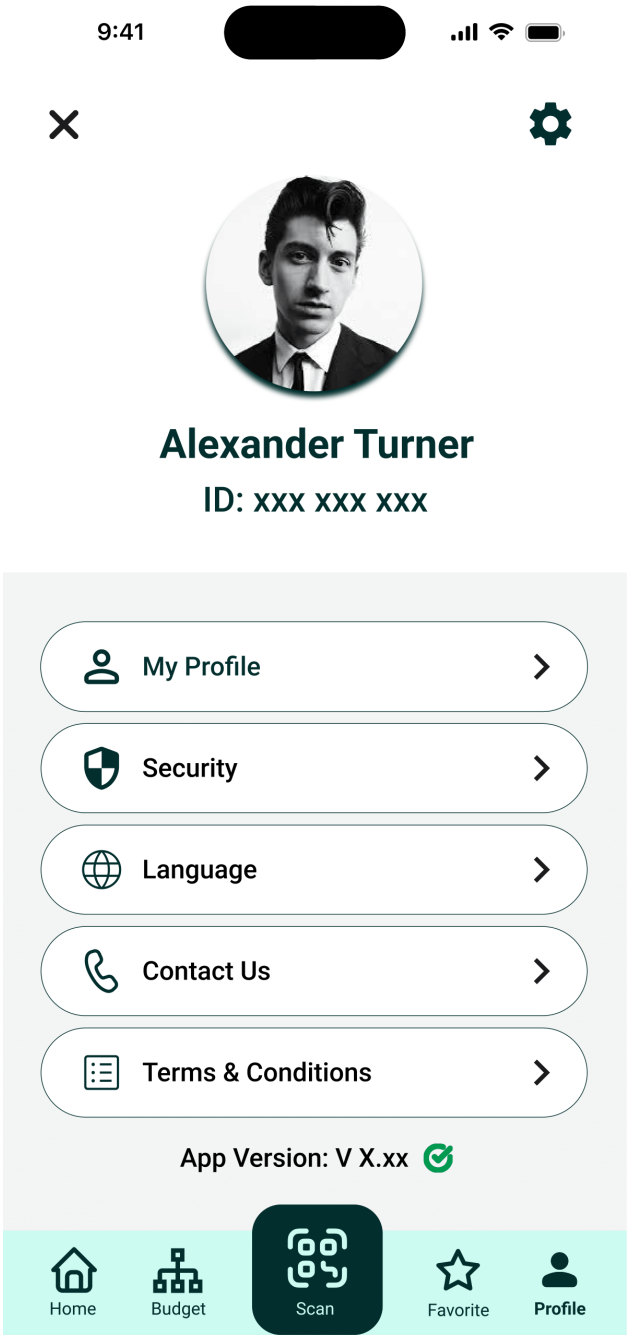
Identifier	Priority	Requirement
REQ-15	5	Users shall be able to see their profile and App Setting Icons 
REQ-16	5	Users shall be able to see their bank net worth, request, and send money. 
REQ-17	5	Users shall be able to see their transaction history and filter. 

Identifier	Priority	Requirement
REQ-18	5	Users shall be able to use the navigation bar to go to different pages. 
REQ-19	5	Users should be able to view details of their transactions with abilities to share, repeat, split, and more. 
REQ-20	5	Users shall be about to view their split payment with other people's contributions. 

Identifier	Priority	Requirement
REQ-21	5	Users can configure budget blocks, set spending limits, and track real-time deductions from each block, with alerts when limits are exceeded. 

Identifier	Priority	Requirement
REQ-22	5	Users can scan or upload QR code to initiate peer-to-peer payments. After scanning, they can choose the budget category and allocate deductions across customized budget blocks and select bank accounts based on predefined percentages. 

Identifier	Priority	Requirement
REQ-23	5	Users shall be able to save other users' bank account details for easier future transfers. 

Identifier	Priority	Requirement
REQ-24	5	Users shall be about to see their profile, ID, check security, Language, Contact Info, and Terms & Conditions. 

Functional Requirement Specification and Use Cases

I. Stakeholders

Having a convenient and secure way to manage multiple bank accounts, track budgets, and make payments can reduce financial stress and improve transparency for users. By simplifying the payment process and eliminating the need for manual calculations, QuickPay helps individuals and groups make smarter financial decisions. This not only benefits users by giving them greater control and clarity but also supports society by encouraging responsible money management and reducing errors or disputes in shared expenses. However, we have identified these primary stakeholders.

A. Primary Stakeholders

Individual Users: Use the application for personal finance management. These users will benefit from linking multiple bank accounts, tracking spending through clear visualizations, and making quick QR code payments. Having transparent, real-time updates about where their money is coming from will reduce confusion and help them stay on top of their financial goals.

Group or Shared Users: Use the application to split expenses with roommates, friends, or family. These users will benefit by avoiding awkward manual calculations and reimbursement requests. The ability to automatically route payments from preferred accounts will make shared financial responsibilities smoother and more reliable.

Developer: Are dedicated to ensuring that QuickPay is reliable, secure, and user-friendly. Their focus on continuous improvement, smooth integration of new features, and regulatory compliance is critical to the success of the application. This commitment ensures that the app delivers lasting value to users and institutions alike.

B. Secondary Stakeholders

Financial Institutions: While they are not directly involved in QuickPay's operations, they benefit from the app's non-custodial design, which keeps user funds within their accounts rather than requiring third-party storage. This reinforces customer trust in their services and encourages continued engagement with their financial product.

II. Actors and Goals

Actor	Participating/ Initiating	Role	Goal
User	**Initiating**	The user interacts with the app to conveniently manage personal finance	Link accounts, view balances, track budgets, and initiate transfers.
Group user	**Initiating**	The group user uses the app to split and settle share expenses	Simplify group payments and avoid manual reimbursement calculations
Plaid API	**Participating**	The Plaid API allows users to link their bank account	Return account, balance, and transaction data safely.
Firebase Database	**Participating**	Stores user profiles, budgeting data, and transaction history	Persist and update financial data reliably in real time.
Stripe Payment Service	**Participating**	Processes bank-to-bank transfers and QR-based payments.	Execute secure, compliant money transfers and deliver funds between users or merchants.
Expo (Framework & Deployment)	**Participating**	Provides libraries and tools for app development and deployment.	Enable fast cross-platform development and support cloud builds and updates for users.
Docker	**Participating**	Provides a containerized environment for development and deployment.	Ensure consistent, portable, and scalable deployment across different systems.

Actor	Participating/ Initiating	Role	Goal
Developer	**Participating**	Design, build, and maintain the application.	Deliver a schedule, reliable, and user-friendly financial platform.

III. Use Cases

A. Causal Description

1. UC-1: Link Multiple Banks

- Description: Allows users to link multiple bank accounts
- Responds to requirements: REQ-1, REQ-4, REQ-5, REQ-7, REQ-10, REQ-12, REQ-13, REQ-16

2. UC-2: Make QR code payment

- Description: Allows users to scan QR code to make payment from user to user
- Responds to requirements: REQ-2, REQ-3, REQ-4, REQ-6, REQ-11, REQ-12, REQ-13, REQ-18, REQ-20, REQ-22

3. UC-3: Group Expense Splitting

- Description: Allows user to split a group expense across multiple accounts.
- Responds to requirements: REQ-2, REQ-6, REQ-11, REQ-12, REQ-13, REQ-20

4. UC-4: Visualize and Manage Budget

- Description: Allow users to see spending and allocations in real time using an interactive tree/flow-chart
- Responds to requirements: REQ-3, REQ-7, REQ-11, REQ-12, REQ-13, REQ-21

5. UC-5: Route Payments Non-Custodially Across Accounts

- Description: Execute payments from users' own accounts without holding funds in the app
- Responds to requirements: REQ-1, REQ-4, REQ-5, REQ-11, REQ-12, REQ-13, REQ-20

6. UC-6: Receive Alerts for Low Balances or Bills

- Description: Provide the user with real-time alerts for important financial events
- Responds to requirements: REQ-7, REQ-12, REQ-13, REQ-24

7. UC-7: View Balances & Manage via Unified Dashboard

- Description: Provide users with a single, unified dashboard where they can view balances across all linked accounts, check recent activity, and perform quick actions without switching between multiple apps.
- Responds to requirements: REQ-1, REQ-5, REQ-12, REQ-13, REQ-15, REQ-16, REQ-21

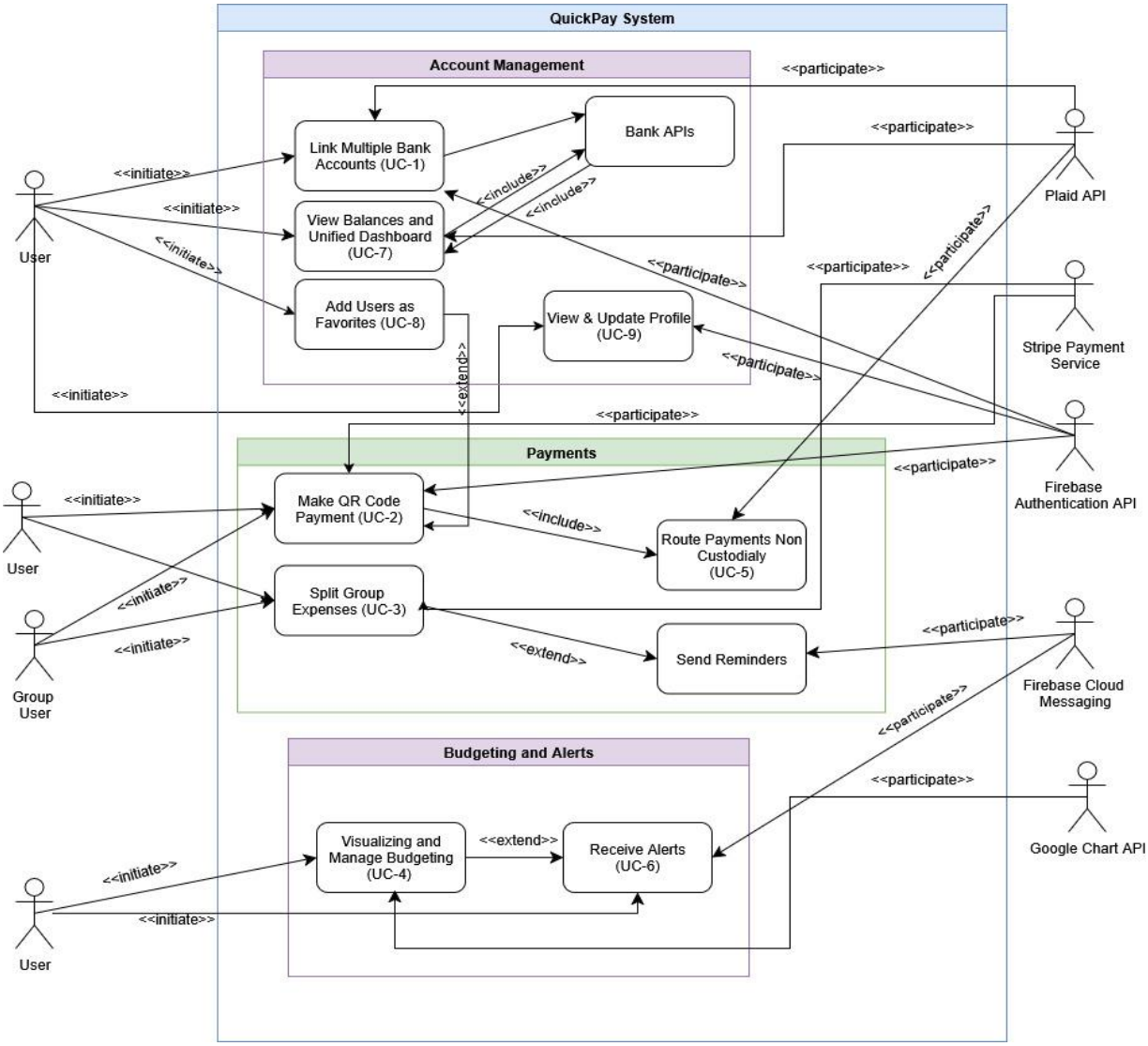
8. UC-8: Add users as favorites

- Description: Let users add other users as their favorites for easy payment.
- Responds to requirements: REQ-8, REQ-12, REQ-13, REQ-23

9. UC-9: Allow users to view and update information in their profile

- Description: Allow users to view and update information in their profile and adjust the app settings.
- Responds to requirements: REQ-9, REQ-12, REQ-13, REQ-24

B. Use Case Diagram



C. Traceability Matrix (1) – System Requirements to Use Cases

REQ	Power	UC-1	UC-2	UC-3	UC-4	UC-5	UC-6	UC-7	UC-8	UC-9
REQ-1	**5**	x				x		x		
REQ-2	**5**		x	x						
REQ-3	**5**		x		x					
REQ-4	**5**	x	x			x				
REQ-5	**5**	x				x		x		
REQ-6	**4**		x	x						
REQ-7	**3**	x			x		x			
REQ-8	**5**								x	
REQ-9	**4**									x
REQ-10	**5**	x								
REQ-11	**3**		x	x	x	x				
REQ-12	**4**	x	x	x	x	x	x	x	x	x
REQ-13	**5**	x	x	x	x	x	x	x	x	x
REQ-14	**5**									
REQ-15	**5**							x		
REQ-16	**5**	x						x		
REQ-17	**5**	x								
REQ-18	**5**		x							
REQ-19	**5**	x		x						
REQ-20	**5**		x	x		x				
REQ-21	**5**				x			x		
REQ-22	**5**		x							
REQ-23	**5**								x	
REQ-24	**5**						x			x
Max Power		**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**	**5**
Total Power		**47**	**46**	**31**	**25**	**32**	**17**	**34**	**19**	**18**

Use Case 1 (Multiple Banks Linking) provides the foundation for account management, allowing users to connect all their bank accounts in one place and set up automatic payments. This use case enables the core functionality that users need to manage their finances without switching between different banking apps.

Use Cases 2 and 5 work together to handle payments, with UC2 enabling QR code scanning for quick transactions and UC5 ensuring payments come directly from user accounts without storing money in the app. These use cases address the main payment needs while keeping user funds secure in their own banks.

Use Cases 3 and 4 focus on collaboration and visualization features. UC3 allows friends and family to easily split shared expenses like restaurant bills or group activities, while UC4 provides interactive budget charts that help users see their spending patterns in real time through visual displays.

Use Cases 6 through 9 enhance the daily user experience by providing helpful notifications for low balances, a unified dashboard to view all account information, favorite contacts for quick payments, and profile management for personal settings. Together, these use cases create a complete financial management system that simplifies money handling for individuals and groups.

D. Fully Dressed Description

Use Case UC-1: Link Multiple Banks

- **Related Requirements:** REQ-1, REQ-4, REQ-5, REQ-7, REQ-10, REQ-12, REQ-13, REQ-16
- **Initiating Actor:** User
- **Participating Actors:** Plaid API, Firebase Database, Developer
- **Preconditions:** User is authenticated via Firebase Authentication and has a stable internet connection; Plaid link flow is available and configured; consent screen and data privacy disclosures are presented
- **Postconditions:** Selected bank accounts are linked to the user profile; account metadata and balances are synced and persisted; dashboard reflects consolidated balances

Use Case UC-1: Link Multiple Banks

- **Flow of Events of Main Success Scenario:**
  1. User opens the unified dashboard and selects Link Bank Accounts
  2. System launches Plaid link flow to fetch bank list and consent
  3. User selects a financial institution and authenticates with the bank
  4. System receives tokens and retrieves account, balance, and transaction metadata
  5. System stores linked to account references and minimal metadata in Firebase and refreshes the dashboard
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Invalid bank credentials: System explains the mismatch and prompts retry or change

  1b. User denies consent: System cancels linking and returns to dashboard without changes

  2a. Plaid API unavailable: System queues a retry and notifies with a non-blocking alert

  3a. Partial account selection: System links only selected accounts and records user preferences

Use Case UC-2: Make QR code payment

- **Related Requirements:** REQ-2, REQ-3, REQ-4, REQ-6, REQ-11, REQ-12, REQ-13, REQ-18, REQ-20, REQ-22
- **Initiating Actor:** User
- **Participating Actors:** Stripe Payment Service, Firebase Database, Plaid
- **Preconditions:** Both sender and recipient have QuickPay accounts; sender has at least one linked funding account; camera or QR image upload is permitted
- **Postconditions:** Payment is authorized and executed; sender transaction history updated; recipient balance state updated; receipt recorded and optionally shareable via link or QR
- **Flow of Events of Main Success Scenario:**
  1. User taps the center QR button to initiate payment
  2. System opens camera scanner and decodes recipient QR payload or accepts uploaded QR image
  3. User enters amount and selects source account or allocation preset if configured

Use Case UC-2: Make QR code payment

4. System requests payment execution through Stripe with non-custodial routing data
5. System confirms success, records the transaction in Firebase, and shows receipt with share options

- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Invalid QR payload: System shows error and allows rescanning or manual recipient selection

  2a. Insufficient funds: System suggests alternate source accounts or split across accounts if enabled

  3a. Network failure: System retrieves gracefully and informs the user of pending status

  3b. Payment timeouts: System cancels authorization and logs a non-finalized attempt

Use Case UC-3: Group Expense Splitting

- **Related Requirements:** REQ-2, REQ-6, REQ-11, REQ-12, REQ-13, REQ-20
- **Initiating Actor:** User Group
- **Participating Actors:** Stripe Payment Service, Firebase Database
- **Preconditions:** An original transaction exists, or a new expense is being created. Group participants are identified or shareable via QR link; notifications are enabled for reminders
- **Postconditions:** Split request is created with per-person amounts; contribution status is tracked; payer receives funds as participants complete payments
- **Flow of Events of Main Success Scenario:**
  1. Group user opens a past transaction or creates a new shared expense and selects Split Payment
  2. System calculates equal or custom shares and generates a QR or share link for participants
  3. Participants scan the QR or open the link and complete their share via supported payment method
  4. System updates contribution status and notifies the originator upon each payment
- **Flow of Events for Extensions (Alternate Scenarios):**

Use Case UC-3: Group Expense Splitting

1a. Custom share ratios: System allows manual adjustments and locks totals to the bill amount

2a. Late or missing contributors: System schedules reminders and provides a status list to the originator

3a. Partial cancellations: System recalculates remaining shares and updates invitations

Use Case UC-4: Visualize and Manage Budget

- **Related Requirements:** REQ-3, REQ-7, REQ-11, REQ-12, REQ-13, REQ-21
- **Initiating Actor:** User
- **Participating Actors:** Firebase Database, Google Charts API
- **Preconditions:** At least one budget block exists, or the user can create new blocks; transactions are synced for categorization; alerts are configured optionally
- **Postconditions:** Budget blocks and limits are saved; visualizations update; alerts for threshold and limit breaches
- **Flow of Events of Main Success Scenario:**
  1. User opens the interactive budgeting page from the navigation bar
  2. System displays total balance, account contributions, and existing budget blocks with limits
  3. User creates or edits blocks, sets limits, and enables alerts for low-balance or overspend events
  4. System stores the configuration, and updates live charts and deduction logic for future transactions
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Reallocate funds: User drags amounts between blocks; system confirms and updates histories

  2a. Category misclassification: User corrects category, and system learns or applies rules for future items

  3a. Alert tuning: User adjusts thresholds or snoozes notifications for a defined period

Use Case UC-5: Route Payments Non-Custodially Across Accounts

- **Related Requirements:** REQ-1, REQ-4, REQ-5, REQ-11, REQ-12, REQ-13, REQ-20
- **Initiating Actor:** User
- **Participating Actors:** Plaid API, Stripe Payment Service, Firebase Database
- **Preconditions:** At least one funding account is linked; allocation presets or percentages may be configured; compliance and consent are satisfied
- **Postconditions:** Payment is executed directly from user accounts without app custody; routing metadata and receipts are recorded; dashboard reflects updated balances
- **Flow of Events of Main Success Scenario:**
  1. User initiates payment and selects Multi-account Allocation
  2. System displays available accounts and saves percentage presets for routing
  3. User confirms percentages or overrides them, then submits for execution
  4. System process payment requests per allocation, receives confirmations, and composes a unified receipt
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. One account fails: System retries or redistributes remaining amount per user approval before finalizing

  2a. Allocation exceeds limit: System suggests a nearest valid allocation within balance and policy constraints

  3a. Compliance hold: System blocks the payment and provides guidance on remediation steps

Use Case UC-6: Receive Alerts for Low Balances or Bills

- **Related Requirements:** REQ-1, REQ-4, REQ-5, REQ-11, REQ-12, REQ-13, REQ-20
- **Initiating Actor:** User
- **Participating Actors:** Firebase Cloud Messaging, Firebase Database, Plaid API

Use Case UC-6: Receive Alerts for Low Balances or Bills

- **Preconditions:** Alerts are enabled with thresholds or schedules; devices have notification permissions
- **Postconditions:** User receives push notifications for configured triggers; alert history is retained; recommended actions may be presented
- **Flow of Events of Main Success Scenario:**
  1. User configures alerts for low balance, unusual activity, and upcoming bills in settings
  2. System monitors linked accounts and budget blocks for threshold crossings and due dates
  3. System sends push notifications with context and quick actions such as transfer or snooze
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. False positive alerts: User marks as not useful and the system tunes sensitivity where applicable

  2a. Device offline: System queues for notifications and delivers when connectivity is restored

Use Case UC-7: View Balances & Manage via Unified Dashboard

- **Related Requirements:** REQ-1, REQ-5, REQ-12, REQ-13, REQ-15, REQ-16, REQ-21
- **Initiating Actor:** User
- **Participating Actors:** Firebase Database, Plaid API
- **Preconditions:** At least one account is linked; the user is authenticated; recent data sync is available
- **Postconditions:** The dashboard shows real-time balances, recent transactions, and quick actions
- **Flow of Events of Main Success Scenario:**
  1. User opens the home page to view current consolidated balance and recent transactions with filters
  2. System presents quick actions for Send, Request, and transaction sharing or splitting

Use Case UC-7: View Balances & Manage via Unified Dashboard

3. User drills into a transaction to view details, repeat a payment, or generate a share link

- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Hidden balance mode: System masks amount until toggled visible

  2a. Bank-specific filter: System shows only selected institution transactions and updates totals

Use Case UC-8: Add users as favorites

- **Related Requirements:** REQ-8, REQ-12, REQ-13, REQ-23
- **Initiating Actor:** User
- **Participating Actors:** Firebase Database
- **Preconditions:** The recipient has a QuickPay account or shareable identifier; user has permission to store contact metadata
- **Postconditions:** Favorite is created with saved identifiers and optional profile image; future payments can target favorites quickly
- **Flow of Events of Main Success Scenario:**
  1. User opens Favorites and selects Add Favorite
  2. System prompts for recipient details or imports from a completed transaction
  3. User saves the favorite and optionally pins it for quick access on the Send flow
- **Flow of Events for Extensions (Alternate Scenarios):**

  1a. Duplicate favorite: System alerts and offers to merge or keep separate with a label

  2a. Recipient updates account: System notifies on next payment attempt and requests confirmation

Use Case UC-9: Allow users to view and update information in their profile

- **Related Requirements:** REQ-9, REQ-12, REQ-13, REQ-24
- **Initiating Actor:** User
- **Participating Actors:** Firebase Authentication, Firebase Database

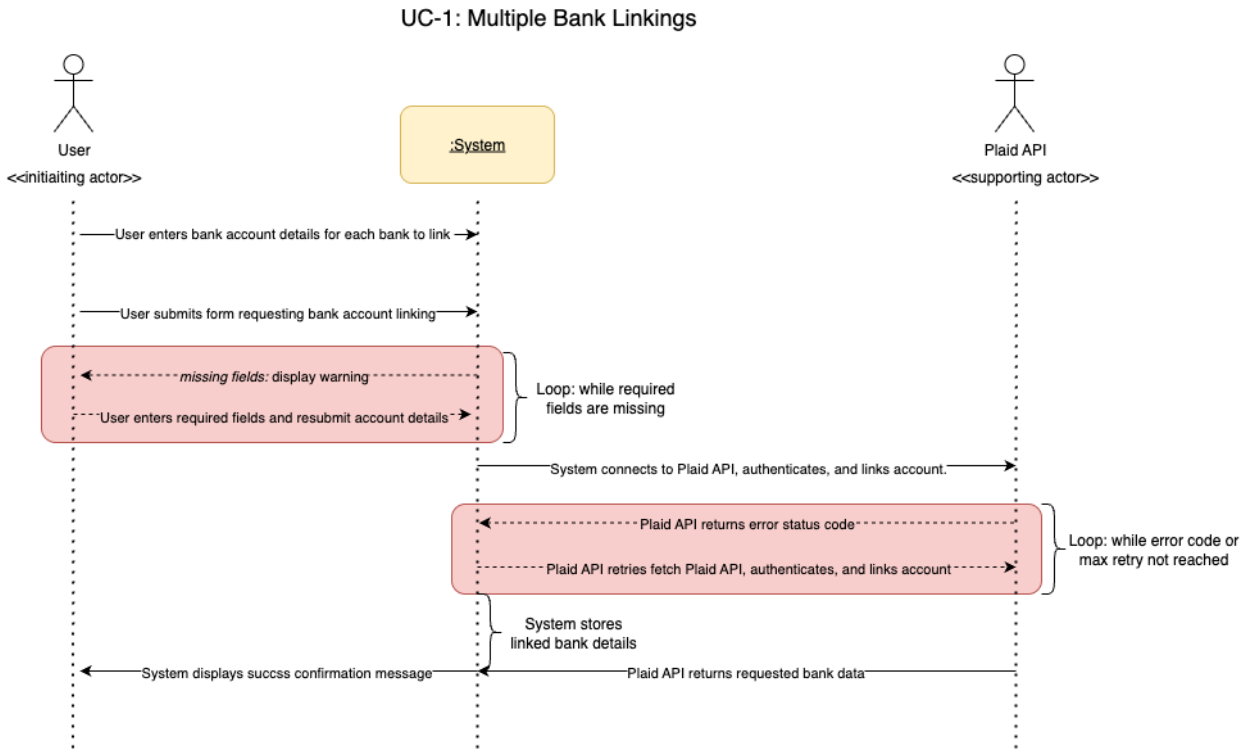
Use Case UC-9: Allow users to view and update information in their profile

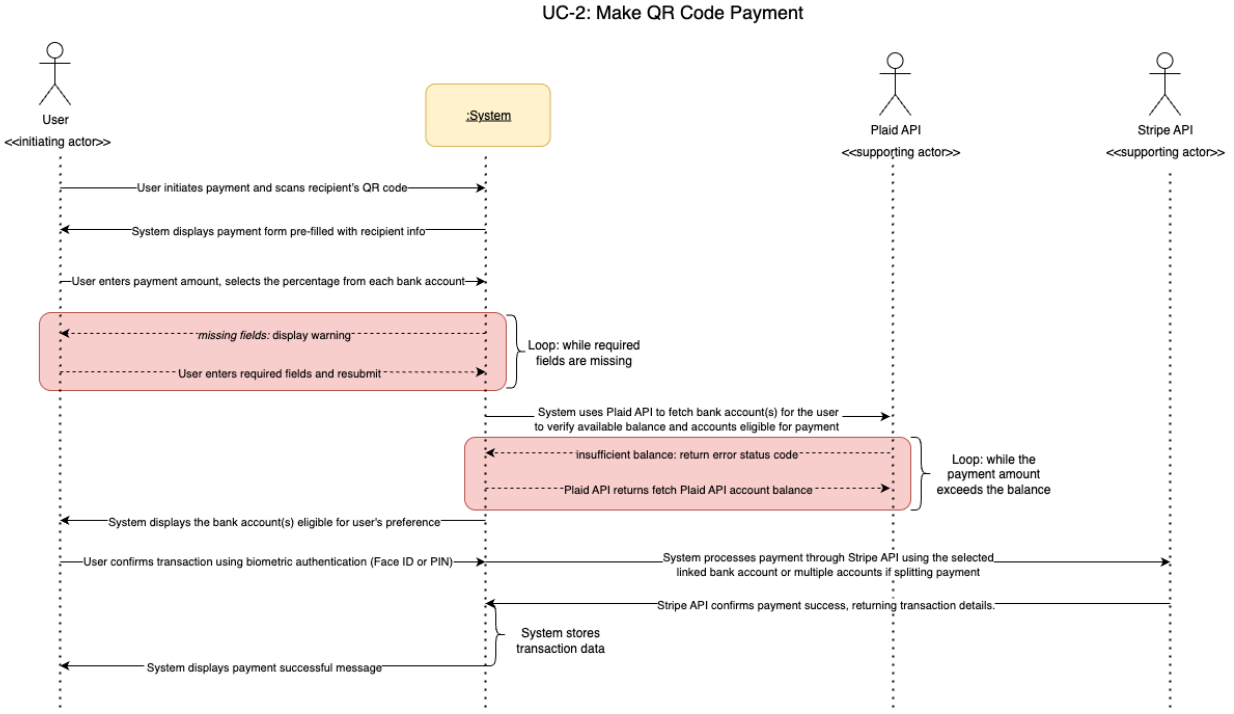
- **Preconditions:** User is authenticated; profile page is accessible; permissions for MFA and language settings are available
- **Postconditions:** Profile details, security settings, language, and terms acceptance state are saved; changes propagate to other features
- **Flow of Events of Main Success Scenario:**
  1. User opens Profile and edits contact information, security options, and preferences
  2. System validates input, updates records, and refreshes the UI state accordingly
  3. User reviews the updated profile and returns to the dashboard
- **Flow of Events for Extensions (Alternate Scenarios):**

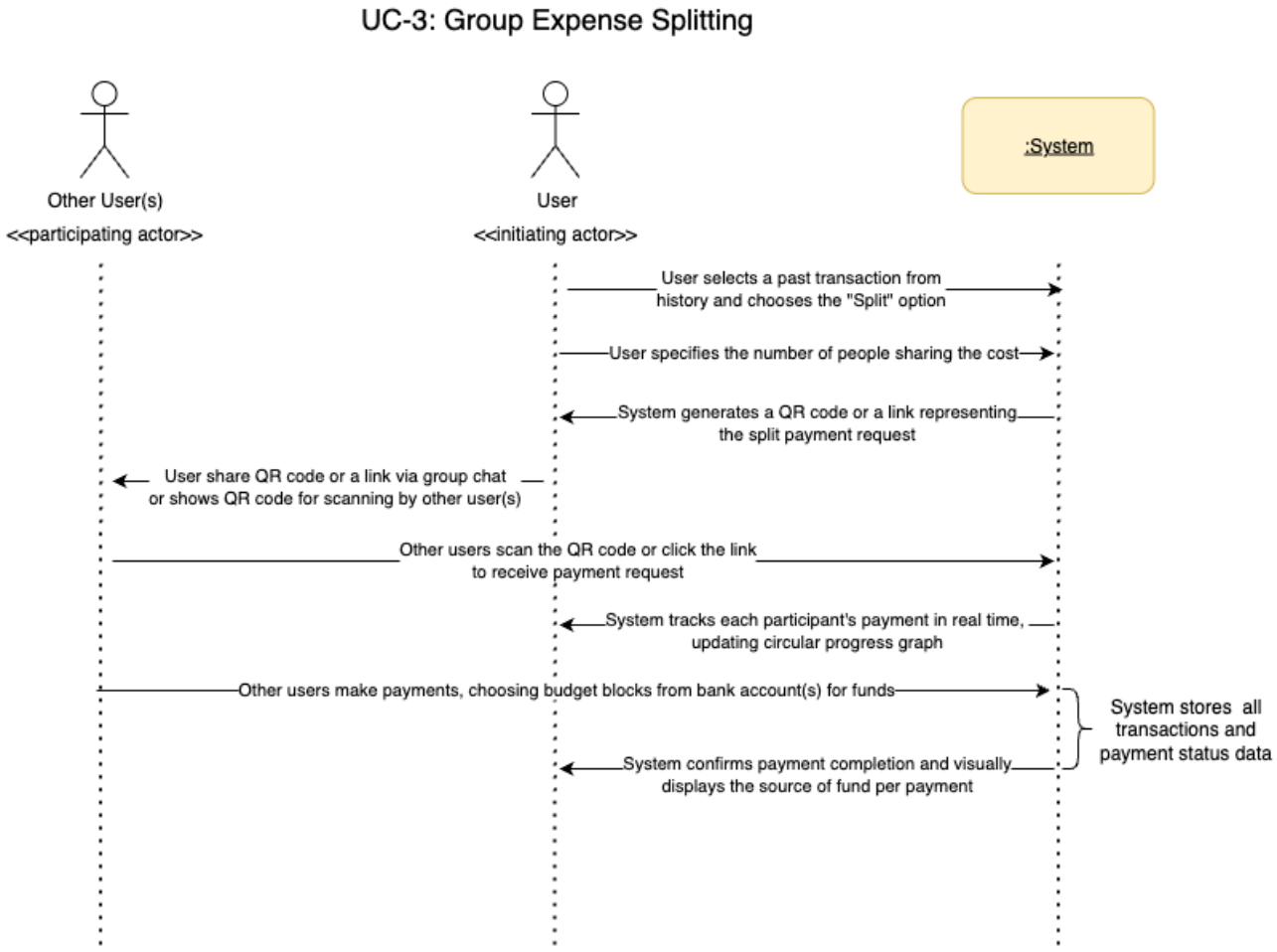
  1a. Weak password or missing MFA: System prompts to strengthen and enable MFA before saving

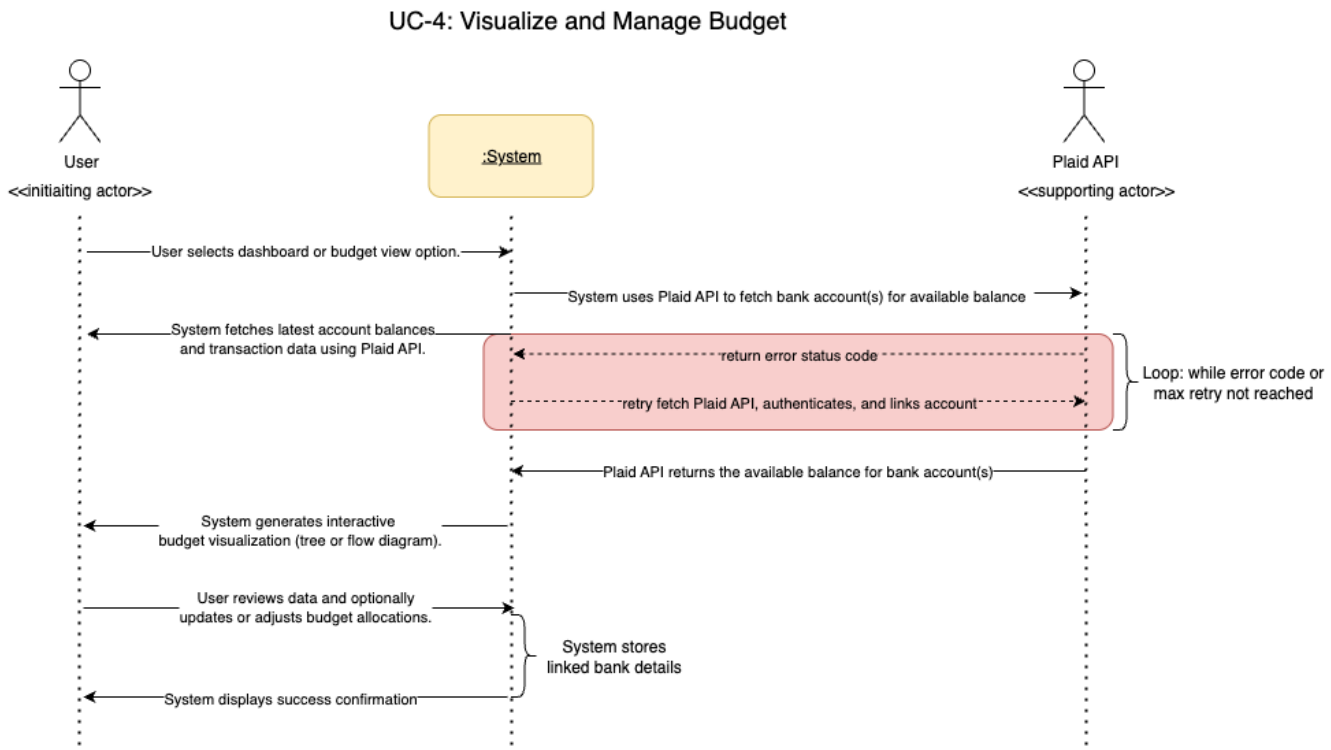
  2a. Data download or delete request: System initiates GDPR or CCPA flow per privacy requirements

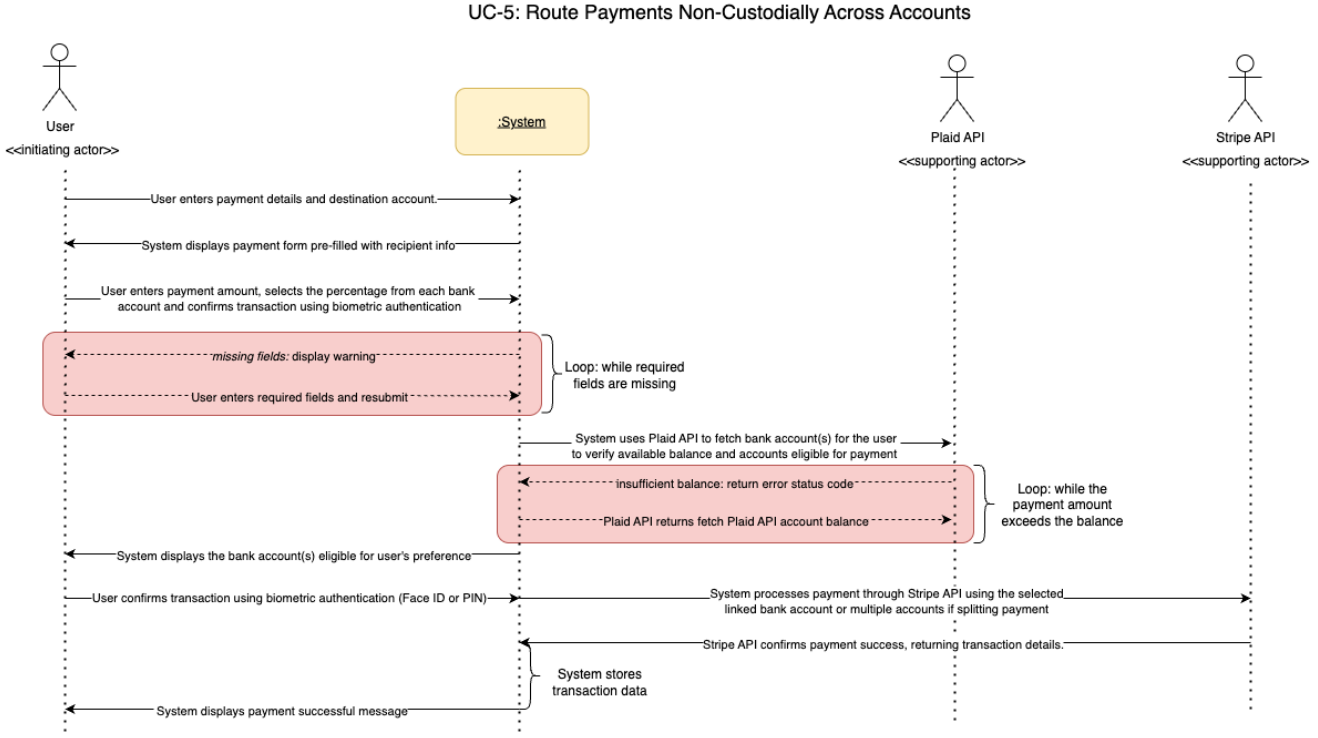
IV. System Sequence Diagrams

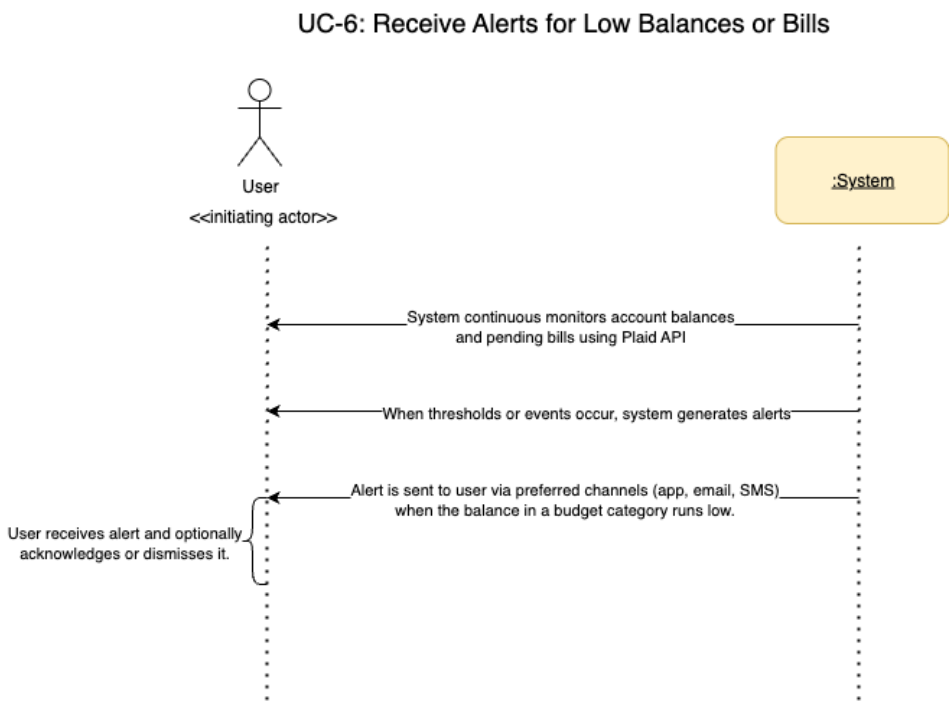


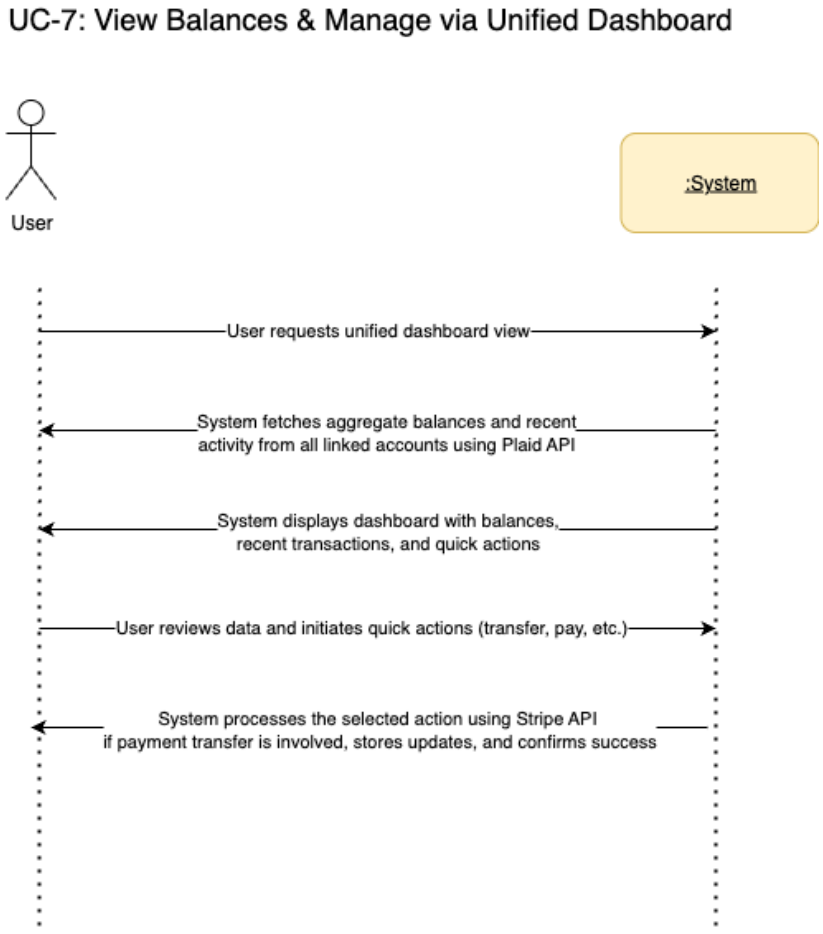


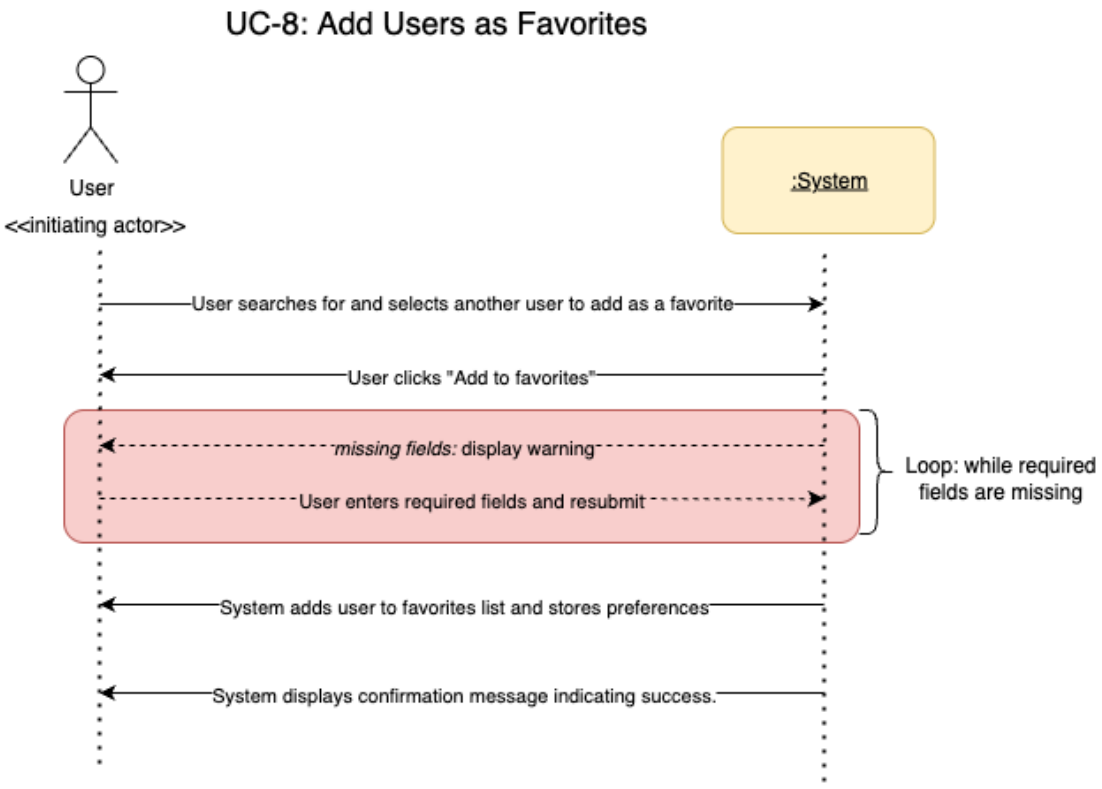

UC-3: Group Expense Splitting
Other User(s)
<<participating actor>>
User
<<initiating actor>>
:System
User selects a past transaction from history and chooses the "Split" option
User specifies the number of people sharing the cost
System generates a QR code or a link representing the split payment request
User share QR code or a link via group chat or shows QR code for scanning by other user(s)
Other users scan the QR code or click the link to receive payment request
System tracks each participant's payment in real time, updating circular progress graph
Other users make payments, choosing budget blocks from bank account(s) for funds
System stores all transactions and payment status data
System confirms payment completion and visually displays the source of fund per payment

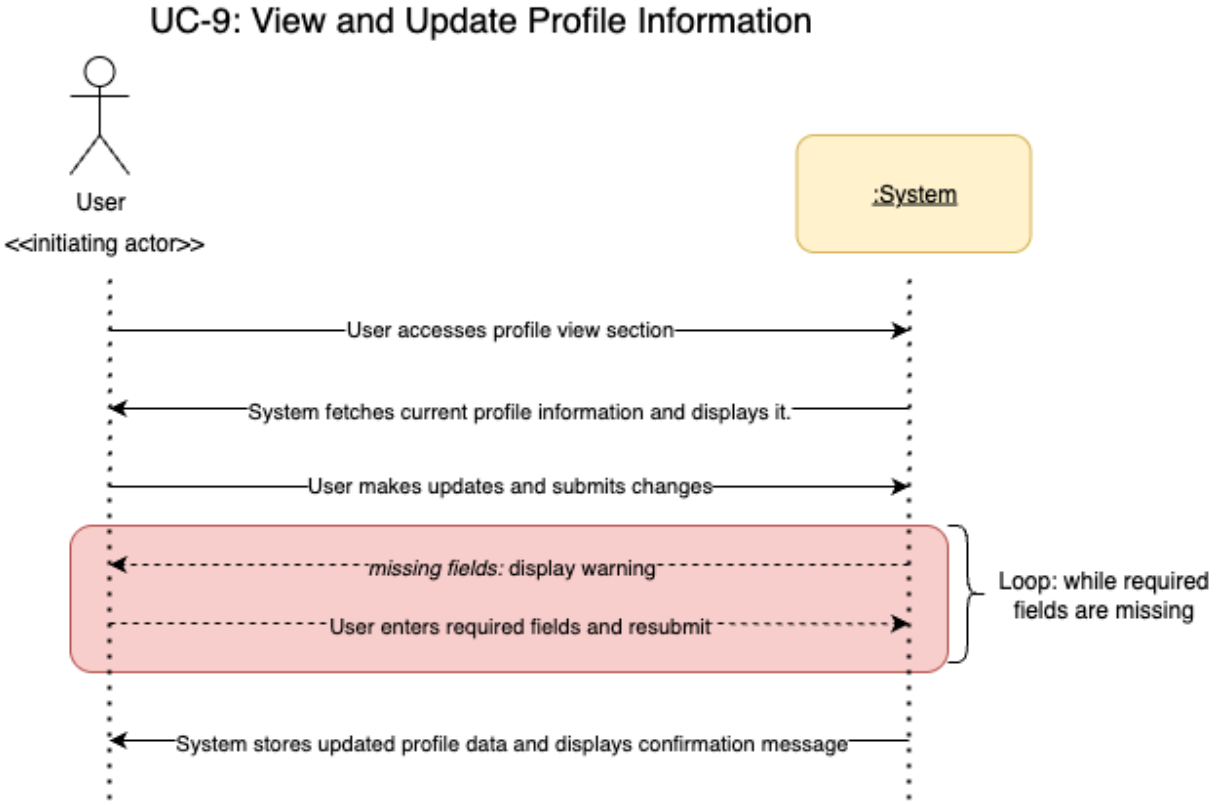

UC-4: Visualize and Manage Budget
User
<<initiaiting actor>>
:System
Plaid API
<<supporting actor>>
User selects dashboard or budget view option.
System uses Plaid API to fetch bank account(s) for available balance
System fetches latest account balances and transaction data using Plaid API.
return error status code
retry fetch Plaid API, authenticates, and links account
Loop: while error code or max retry not reached
Plaid API returns the available balance for bank account(s)
System generates interactive budget visualization (tree or flow diagram).
User reviews data and optionally updates or adjusts budget allocations.
System stores linked bank details
System displays success confirmation


UC-5: Route Payments Non-Custodially Across Accounts
User
<<initiating actor>>
:System
Plaid API
<<supporting actor>>
Stripe API
<<supporting actor>>
User enters payment details and destination account.
System displays payment form pre-filled with recipient info
User enters payment amount, selects the percentage from each bank account and confirms transaction using biometric authentication
missing fields: display warning
User enters required fields and resubmit
Loop: while required fields are missing
System uses Plaid API to fetch bank account(s) for the user to verify available balance and accounts eligible for payment
insufficient balance: return error status code
Plaid API returns fetch Plaid API account balance
Loop: while the payment amount exceeds the balance
System displays the bank account(s) eligible for user's preference
User confirms transaction using biometric authentication (Face ID or PIN)
System processes payment through Stripe API using the selected linked bank account or multiple accounts if splitting payment
Stripe API confirms payment success, returning transaction details.
System stores transaction data
System displays payment successful message


UC-6: Receive Alerts for Low Balances or Bills
User
<<initiating actor>>
:System
System continuous monitors account balances and pending bills using Plaid API
When thresholds or events occur, system generates alerts
Alert is sent to user via preferred channels (app, email, SMS) when the balance in a budget category runs low.
User receives alert and optionally acknowledges or dismisses it.


UC-7: View Balances & Manage via Unified Dashboard
User
:System
User requests unified dashboard view
System fetches aggregate balances and recent activity from all linked accounts using Plaid API
System displays dashboard with balances, recent transactions, and quick actions
User reviews data and initiates quick actions (transfer, pay, etc.)
System processes the selected action using Stripe API if payment transfer is involved, stores updates, and confirms success


UC-8: Add Users as Favorites
User
<<initiating actor>>
:System
User searches for and selects another user to add as a favorite
User clicks "Add to favorites"
missing fields: display warning
User enters required fields and resubmit
Loop: while required fields are missing
System adds user to favorites list and stores preferences
System displays confirmation message indicating success.


UC-9: View and Update Profile Information
User
<<initiating actor>>
:System
User accesses profile view section
System fetches current profile information and displays it.
User makes updates and submits changes
missing fields: display warning
User enters required fields and resubmit
Loop: while required fields are missing
System stores updated profile data and displays confirmation message

User Interface Specification

I. Preliminary Design

A. Navigation Bar

1.1. The **first button in the navigation bar**, represented by a home icon, serves as the default page where users can view their current balance and recent transactions.

1.2. The **second button**, **with a** Flowchart icon, opens the interactive budget view, allowing users to see all linked bank accounts and their real-time financial standing.

1.3. The **center dark teal QR code button** in the navigation bar provides a quick shortcut for instant payment scanning.

1.4. The **fourth button**, with a **Star icon**, displays the user's saved favorite accounts for faster transactions.

1.5. The **fifth button**, with a **Person icon**, opens the user profile and general settings.

B. Home Page

1.6. Users can view their verified profile, which they may choose to share with others when transferring money. Unverified users will not be able to view other users' profiles during transactions.

1.7. The profile and settings icons at the top-right corner allow users to update their profile, configure security settings, change the language, or review the terms & conditions.

1.8. Users can see their current balance across all connected bank accounts in QuickPay, with the option to show or hide the balance.

1.9. The "Request" button lets users receive money by sharing either a QR code or a payment link.

1.10. The "Send" button allows users to transfer money to another QuickPay user by entering their account number manually or scanning a QR code.

1.11. Transactions Section:

1.12. Users can filter recent transactions across all linked bank accounts, view transactions from a specific bank, or isolate shared (split) payments and personal transactions.

1.13. Incoming transactions are displayed as green positive values, while outgoing transactions are shown as red negative values on the right side.

1.14. Each transaction can be expanded to access a Share button, allowing users to generate a link or QR code to request a split payment from multiple users or share the transaction receipt.

1.15. Users can **repeat a transaction** to send a payment to the same recipient.

1.16. The pie chart icon provides access to the split payment history for all payments received from any account where a split payment was requested.

1.17. The three-dot icon displays detailed information about a transaction, including the exact date and other relevant details.

C. Interactive Budgeting Page

1.18. Users can visualize their financial overview through a pie chart displaying contributions from each connected bank account, view their total balance across all accounts, and see which banks are linked to their QuickPay balance.

1.19. Users can access an interactive "playground" interface to manage their budget blocks, which branch from their main balance and are customized with names reflecting their spending categories (wants, needs, general expenses, etc.).

1.20. Users can monitor fund allocation across budget blocks to track their financial position and transfer money between blocks, with real-time deduction of payments from the appropriate budget categories.

1.21. Users can expand individual budget blocks for detailed views, including allocation breakdowns, spending limits, and transaction histories specific to each budget category.

D. QR Code Scanning Page

1.22. Users can activate their mobile camera and position the QR code within the designated scanning frame.

1.23. Users can turn on the flash function to improve QR code capture in low-light conditions under the scanning frame.

1.24. Users can upload a QR code image from their device using the upload button located next to the flashlight control.

E. Favorites Page

1.25. Users can view all their saved accounts for faster transactions, with profile images displayed for contacts who have chosen to share their photos.

F. Profile Page

1.26. Users can view their profile picture along with a list of options for general and security settings. Contact and Terms and Conditions buttons are also available.

II. User Effort Estimation

A. User Scenario 1: Effortless Multi-Account Payments with Smart Allocation

1. Task A: Setting Up Automatic Rent Payment with Custom Allocation

Steps:

1. Navigate to home page (tap: 1)
2. Tap "Send" button (tap: 1)
3. Enter recipient details or scan QR (keystrokes: ~15 or taps: 3)
4. Enter payment amount (keystrokes: ~6)
5. Access multi-account allocation settings (tap: 2)
6. Set percentage for Account 1 (keystrokes: ~3)
7. Set percentage for Account 2 (keystrokes: ~3)
8. Set percentage for Account 3 (keystrokes: ~3)
9. Enable recurring payment (tap: 1)
10. Set payment schedule/date (taps: 3)

Confirm and save (tap: 1)

Total Efforts: 12 taps + 30 keystrokes = 42 actions

1.27. **Navigation:** 12 actions (29%)

1.28. **Data Entry:** 30 actions (71%)

2. Task B: Viewing Payment Status and History

Steps:

1. Open app to home page (tap: 1)
2. Scroll to transactions section (swipe: 1)
3. Tap on specific transaction (tap: 1)
4. View allocation breakdown (tap: 1)

Total Efforts: 3 taps + 1 swipe = 4 actions

1.29. **Navigation:** 3 taps + 1 swipe = 4 actions (100%)

1.30. **Data Entry:** 0 action (0%)

B. User Scenario 2: Interactive Visual Budgeting for Smart Spending

1. Task A: Setting Up Budget Category and Limits (Single Category)

Steps:

1. Tap flowchart icon in navigation (tap: 1)
2. Create "Shopping" budget block (tap: 2)
3. Name the category (keystrokes: ~8)
4. Set budget limit (keystrokes: ~6)
5. Save configuration (tap: 1)
6. View allocation breakdown (tap: 1)

Total Efforts: 4 taps + 14 keystrokes = 18 actions

1.31. **Navigation:** 4 actions (22%)

1.32. **Data Entry:** 14 actions (78%)

2. Task B: Monitoring and Reallocating Funds

Steps:

1. Tap flowchart icon (tap: 1)
2. Tap on budget block to expand (tap: 1)
3. View current allocation and spending (no action)
4. Drag funds between blocks (drag gestures: 3)
5. Confirm reallocation (tap: 1)

Total Effort: 3 taps + 3 drags = 6 actions

1.33. **Navigation:** 6 actions (100%)

1.34. **Data Entry:** 0 actions (0%)

C. User Scenario 3: Instant Bill Splitting with QR Payments

1. Task A: Creating a Split Payment from Transaction History

Steps:

1. Open app to home page (tap: 1)
2. Scroll to transactions section (swipe: 1)
3. Tap on grocery transaction (tap: 1)

4. Tap "Share" button (tap: 1)
5. Select "Split Payment" option (tap: 1)
6. Enter number of people sharing (keystrokes: ~1)
7. Generate QR code (tap: 1)
8. Share QR code via group chat (tap: 2)

Total Effort: 7 taps + 1 swipe + 1 keystroke = 9 actions

1.35. **Navigation:** 8 actions (89%)

1.36. **Data Entry:** 1 action (11%)

2. Task B: Making Direct Peer-to-Peer Payment

Steps:

1. Tap center QR button or "Send" (tap: 1)
2. Scan friend's QR code or select from favorites (tap: 1)
3. Enter amount (keystrokes: ~5)
4. Select source account(s) (tap: 2)
5. Confirm payment (tap: 1)

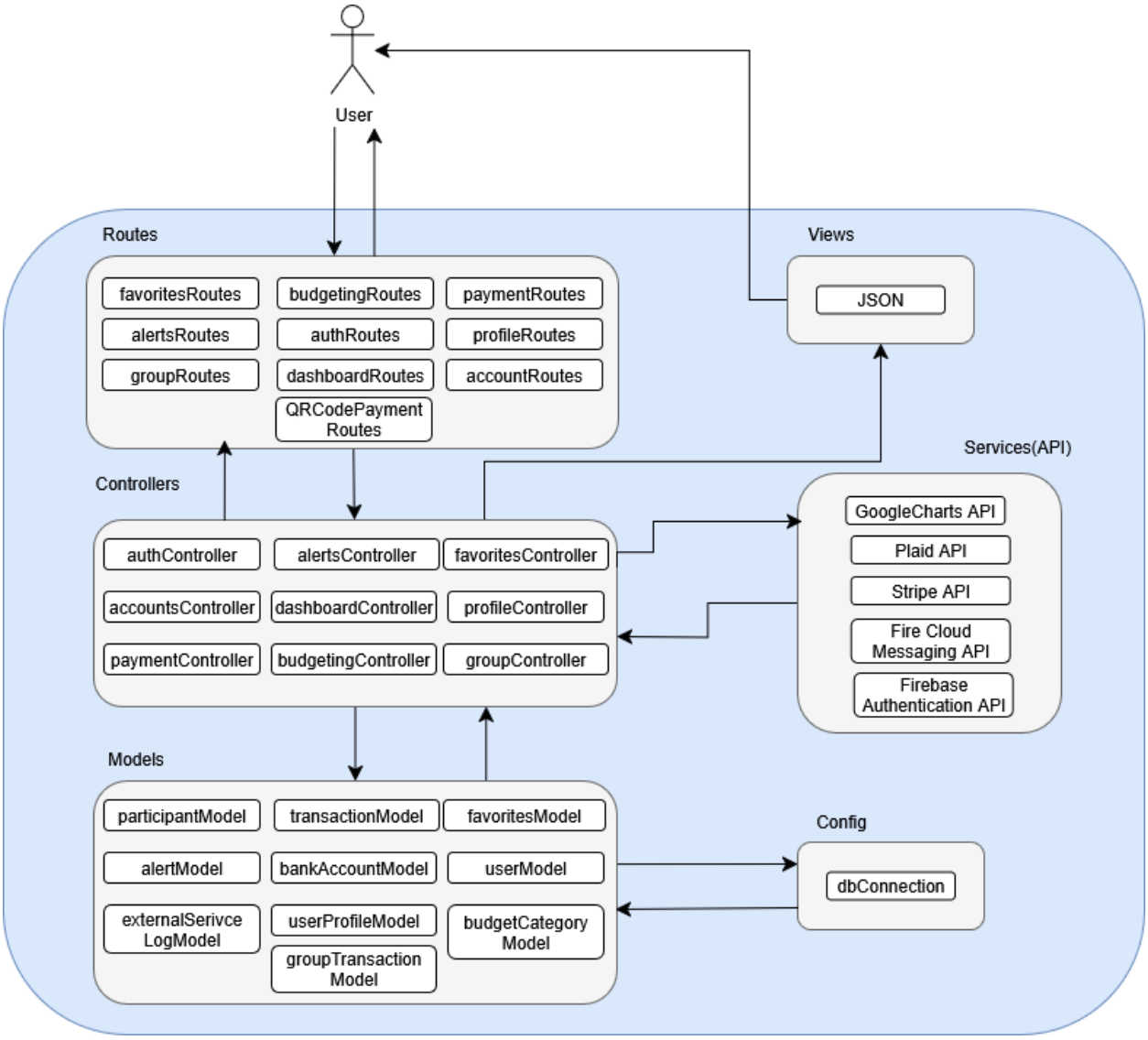
Total Effort: 5 taps + 5 keystrokes = 10 actions

1.37. **Navigation:** 5 actions (50%)

1.38. **Data Entry:** 5 actions (50%)

System Architecture and System Design

I. Identifying Subsystems



The system follows MVC design pattern and consists of the following subsystems: models, controllers, routes, views, config and services.

A. Routes

The subsystem Routes serves as entry points for User request such as making payments, accessing favorites or dashboard and forwarding them to the appropriate Controllers for processing.

Basically, Routes provides a structure for request handling without containing business logic themselves.

B. Controllers

The controllers serve as the core of the QuickPay backend system. It orchestrates application logic and user interactions. The controllers receive API requests through Routes, and it validates and authorizes inputs and invokes Models for database operations while also interacting with services to handle external APIs (Plaid, Stripe, Firebase Auth, Firebase Cloud Message, Google Charts). Overall, the controller assembles the results and generates JSON views for the User to interact with.

C. Models

This subsystem is the data access layer of the application. It follows the CRUD operation (Create, Read, Update, Delete) against the SQL database through the shared database connection, and interacts with Firebase services. The models represent core entities such as Users, Bank Accounts, Transaction, Budget, Alerts, Favorites and Group Transaction. The models are invoked by Controllers to perform data retrieval and manipulation which ensures persistence, consistency and easy integration with both relational data and Fire-based managed database and notifications.

D. Services

The QuickPay Services subsystem contains adapters that connect the backend with external systems. Basically, it integrates API such as Plaid API (bank linking and balances), Stripe API (payment processing), Firebase Authentication (identity), Firebase Cloud Messaging (push notifications), and Google Charts (analytics/visualization support).

E. Config

This subsystem manages configuration and infrastructure settings such as database connection, API keys and Firebase setup. Config ensures all setups are consistent and initialized correctly and can work smoothly.

F. Views

In QuickPay, Views represent the JSON responses returned to the mobile User After Controllers process requests and gather results from Models or Services, they format the output into structured JSON payloads. These responses are then processed by the React Native application, which handles rendering and user interaction on the User side.

II. Architecture Styles

We chose MVC(Model-View-Controller) architecture style for our application. MVC is a good choice for mobile applications that are also API based since it offers flexibility and scalability.

In QuickPay, the MVC architecture style each and subsystems have its own responsibility. Models handle CRUD operations on the SQL database while also integrating with Firebase services for authentication and notifications. Controllers orchestrate application logic calling Models for data access and Services for APIs such as Plaid API, Stripe API, Firebase and Google Charts API. Views are in JSON format as it is processed by React Native Client App.

QuickPay also follows a clint-service architecture, since it communicates with Node.JS/Express backend over secure HTTPS. It processes requests, applies business logic, queries the SQL database, and interacts with external providers. By decoupling the frontend UI from backend logic, the system ensures modularity and makes it possible to expand features without redesigning the entire application.

Furthermore, QuickPay uses an event-driven design for alerts. Domain events set off alerts (such low balances, bill reminders, or group expense changes). To alert users in real time, controllers bundle these events and send them using Firebase Cloud Messaging. This makes it possible to process data asynchronously and send important financial data on time.

QuickPay creates a reliable and expandable architecture by fusing client-server, MVC, and event-driven concepts. This method supports long-term maintainability and scalability by enabling the addition of new features (such as more payment sources, new budgeting visualizations, or extended alert types) with little alteration to the current code.

III. Mapping Subsystems to Hardware

A. Database Subsystem

This subsystem manages persistent storage of user data, linked bank accounts, transactions, budgets, alerts, favorites, and group expenses. We host our database on a cloud server Firebase and Firestore for additional profile and real-tune data handling.

B. Mobile Client Subsystem

This subsystem runs on the user's smartphone (IOS) by the react native applications. It displays JSON response from the backend as the user interface handles QR code scanning and manages the User's budget.

C. Backend Application Subsystem

This subsystem runs on a cloud server. It hosts Routes, Controllers. Models, and Config to process requests from mobile users, validate **users** and execute business logic.

D. Payment and Banking API Interfaces

This subsystem connects API to operate financial transactions. It uses Plaid API to link and retrieve account balances, and with Stripe API to handle payments.

E. Authentication and Notification Services

This subsystem leverages Firebase Authentication for secure user login and identity authentication while Firebase Cloud Messaging is used for sending notifications to Users about alerts, low balances reminder, etc.

F. Analytics & Visualization Subsystem

This subsystem uses Google Charts API to generate budget visualizations and spending flow diagrams. The backend fetches and formats the data, which is returned to the mobile app as JSON for rendering.

IV. Connectors and Network Protocols

A. HTTPS

All communication between the QuickPay mobile client and the backend uses the HTTPS protocol to ensure secure, encrypted data transfer.

B. API Connectors

QuickPay integrates with several external services through API connectors. These include Plaid API, Stripe API, Firebase Authentication, and Google Charts. These connectors handle API authentication, requests, and responses, all transmitted securely over HTTPS.

C. Notification Center

QuickPay uses Firebase Cloud Messaging (FCM) as its notification connector. This enables the system to deliver real-time push notifications to users, such as low-balance alerts, bill reminders, or group expense updates

V. Global Control Flow

A. Execution Orderliness

Our system uses event-driven architecture, the backend waits for User's events such as logging in, linking accounts or making payments and System's events such as over budgeting reminders and upcoming bills alert. The events trigger the controller that processes the events by calling the models or services to return to the User as JSON responses.

B. Time Dependency

Our system has important timing requirements such as reminders, low balance notifications, etc. Scheduled checks run in the background to detect when our user's bank account balances are running low or delayed group expense payments. When these conditions are met, the system triggers alert events, which are processed and dispatched to users through Firebase Cloud Messaging (FCM).

VI. Hardware Requirements

A. Mobile Screen Display

Our system is for mobile applications only and is built on React Native. It's designed to run on both IOS and Android, so it requires support for various screen sizes between each smartphone.

B. Communication Network

QuickPay requires a stable and secure internet connection to maintain real-time synchronization between the mobile client, backend server, SQL database, and Firebase services.

C. Application Server

The backend subsystem (Node.js/Express) will be deployed on a cloud-hosted server (e.g., AWS, Azure, or Google Cloud).

D. Database Server

The database subsystem uses Firebase Fire store, a cloud-hosted SQL database managed on Google Cloud infrastructure.

E. Firebase Services

QuickPay leverages Firebase Authentication, Firestore, and Cloud Messaging, all hosted on Google Cloud infrastructure.

Project Management and Plan of Work

I. Project Development Progress

A. Merging Contributions from Individual Team Members

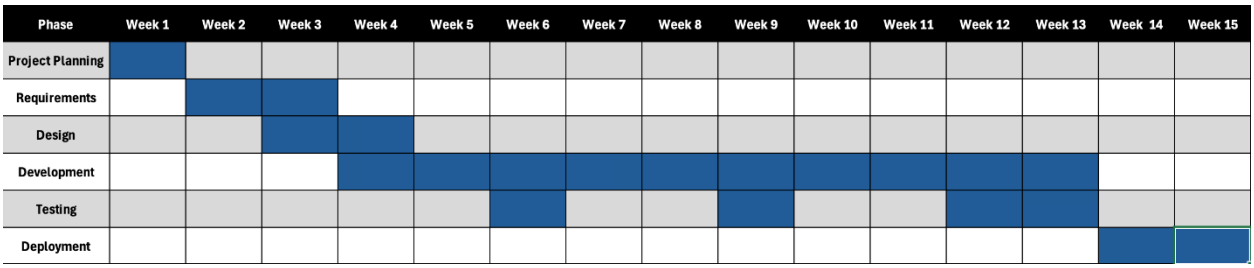
To ensure smooth project progress, our group holds in-person meetings daily from 7:00 to 9:30 PM, Sunday through Thursday, and from 3:30 to 6:00 PM on Saturdays. This meeting enabled us to consistently be in sync with each other on which path the project should take. During the first week we were able to set up our shared documents such as Microsoft word share point, GitHub repo and Gitlab repo. Other than shared documents we were able to come to agreement on our project's framework, API, database system and functionality.

We were able to make projects flow smoothly with our shared documents that we created with Microsoft word share point. We prefer to use Microsoft word share point instead of googling docs because Microsoft word share point has better formatting features, flexibility and better table formatting.

With Microsoft Words share point we were able to work collaboratively while also being able to review each other's work. At the end of each meeting each member will review each section and note down suggested changes to talk about at the next meeting. Even with the group working well with each other and being able to come to an agreement with the concepts, logic and design behind our system and report sections, we met some hiccups such as the formatting of our documents such as headings, and formatting inconsistencies with our writing style. The final step of our project should be reviewing the documents to make sure that the headings, tables and each section fit Dr Michael's standards for a project report.

II. Plan of Work

A. Original Project Timeline



B. Actual Work Timeline

Task Name	Plan Duration	Actual Start	Actual End	Actual Duration
Requirements and Documentation				
Project Proposal	7	08/25/25	08/29/25	5
Report #1 Part 1	7	09/03/25	09/06/25	4
Report #1 Part 2	7	09/08/25	09/14/25	7
Report #1 Part 3 - Full	7	09/15/25	09/21/25	7
Design				
UI/UX Design on Figma	7	09/03/25	09/06/25	4
Development				
Frontend Development on Expo	30	09/10/25		
Testing				
Deployment				

III. Project Report Progress

In the second week, our group worked on the project proposal. This included writing the problem statement, project goals, and system ideas. The proposal gave us the same understanding and helped set a clear direction for the project.

In the third week, we started Report 1, Part 1. This part includes the cover page, table of contents, work assignment, and customer problem statement. Each member worked on their own section, and then we reviewed together to keep the format and style the same.

In the fourth week, we completed Report 1, Part 2. This part includes the functional requirement specification with use cases, actors and goals, use case diagrams, traceability matrix, fully dressed descriptions, system sequence diagrams, and user interface specifications. We continued our collaborative approach with each member contributing to different sections and conducting thorough reviews to maintain consistency.

In the fifth week, we completed Report 1, Part 3. This part focused on the system architecture, subsystem identification, connectors and network protocols, hardware requirements, and project management planning. Similar to our earlier process, we divided responsibilities among members and then conducted collective reviews to ensure accuracy, consistency, and alignment with our overall project direction.

So far, we have successfully completed the proposal, Report 1 Part 1, Part 2, and Part 3. Our systematic approach of dividing work among team members followed by collective reviews has proven effective in maintaining quality and consistency across all deliverables. We are now ready to proceed to the next phase of our project development.

References

1. Plaid API: https://plaid.com/docs/api/
2. Firebase Authentication API: https://firebase.google.com/docs/reference/rest/auth
3. Firebase Cloud Messaging API: https://firebase.google.com/docs/reference/fcm/rest
4. Google Charts API: https://developers.google.com/chart/interactive/docs/quick_start