Team 02 - M2 Technical Documentation

Title Page

SW Engineering CSC648-848-05 Summer 2025

Limóney (Financial Budgeting WebApp)

Team 02

Milestone 2

7/3/25

Name	Role
Emily Perez (eperez@sfsu.edu)	Team Lead
Ishaank Zalpuri	Database Administrator
Andrew Brockenborough	Technical Writer
Dani Luna	Github Master, Database Administrator, Frontend Lead
Jonathan Gonzalez	Software Architect
Gene Orias	Backend Lead

Milestone	Version	Date
Milestone 2	Version 1	7/3/25
Milestone 1	Version 2	7/2/25
Milestone 1	Version 1	6/17/25

Table of Contents

Data Definitions

1. User

- **userID** (INT, PK): Unique identifier for each user.
- **name** (VARCHAR(40)): Full name of the user.
- **email** (VARCHAR(40)): User's email address (must be unique and validated on registration).
- **password** (VARCHAR(255)): Encrypted password using bcrypt or equivalent hashing.
- **user_type** (ENUM: standard, admin): Determines access privileges and UI experience.

Notes:

- Admin users have access to additional interfaces for user and support management.
- **Email** is used as the main login credential and for password recovery.
- Passwords are never stored in plain text.

2. Account

- **accountID** (INT, PK): Unique account ID.
- **userID** (INT, FK): Links the account to a specific user.
- **balance** (INT): Current balance in cents (to avoid float precision issues).
- **account_type** (ENUM: checking, savings, credit): Used to control behavior and visuals in the UI.

3. Transaction

- **transactionID** (INT, PK): Unique ID for each transaction.
- **accountID** (INT, FK): Linked account.
- **amount** (INT)
- **type** (ENUM: income, expense, transfer): Defines the flow of money.

- **category** (VARCHAR(40)): User-defined or system-assigned category.
- **date** (DATE): Date of the transaction.

4. Receipt (BLOB Media)

- **receiptID** (INT, PK): Unique identifier.
- **transactionID** (INT, FK): The transaction it belongs to.
- **date_uploaded** (DATE): Upload date.
- **image** (BLOB): Binary image file (max 5MB, JPG/PNG only).

Notes:

- Stored as BLOB in the database for tight integration with transactions.
- Will enforce max upload size and allowed MIME types on backend validation.

5. ReimbursementRequest

- **requestID** (INT, PK): Unique request ID.
- **transactionID** (INT, FK): Associated transaction.
- **status** (ENUM: pending, approved, rejected): Workflow state for admin review.

6. Budget

- **budgetID** (INT, PK)
- **accountID** (INT, FK)
- **limit_amount** (INT): Stored in cents.
- **start_date** / **end_date** (DATE): Defines the budget period.

7. Subscription

- **subscriptionID** (INT, PK)
- **accountID** (INT, FK)
- **name** (VARCHAR(40)): Name of the service (e.g., Netflix).
- **amount** (INT): Per interval, in cents.
- **interval** (ENUM: daily, weekly, monthly, yearly)

- **next_due_date** (DATE): For notifications.

8. Task

- **taskID** (INT, PK)
- **userID** (INT, FK)
- **type** (ENUM: daily, weekly, one-timer, custom)
- **status** (ENUM: pending, completed, overdue)

9. Reward

- **rewardID** (INT, PK)
- **userID** (INT, FK)
- **points** (INT): Earned through task completions, budget success, etc.

10. Notification

- **notificationID** (INT, PK)
- **userID** (INT, FK)
- **type** (ENUM: info, alert, reminder)
- **content** (TEXT): Message content.
- **date** (DATETIME): When the notification is sent.

11. LemonAidLogs (AI Interaction Log)

- **logID** (INT, PK)
- **userID** (INT, FK)
- **output** (TEXT): LemonAid's response.
- **timestamp** (TIMESTAMP): When the AI response was logged.
- **rating** (ENUM: 1–5): User-provided feedback on AI quality.

***Notes*:**

- A unique and competitive feature of the app, this log helps track AI usefulness and identify areas for improvement.

- May later support feedback-based learning or analytics.

12. SupportTicket

- **ticketID** (INT, PK)
- **userID** (INT, FK): Creator of the ticket.
- **subject** (VARCHAR(40))
- **message** (TEXT)
- **status** (ENUM: open, in progress, closed)
- **admin_response** (TEXT): Optional reply from admin.

13. Administrator

- **adminID** (INT, PK, FK from User): Inherits from User.
- **permissions** (TEXT): Lists actions admin is allowed to perform

14. AccountBankLink (Associative Table)

- **accountID** (INT, FK)
- **bankID** (INT, FK)

User Privileges and Registration Info

- **Standard users** can:
  - Register, login, and manage their accounts, tasks, budgets, and AI interactions.
  - View only their own data.
- **Admins** can:
  - View user accounts, support tickets, and analytics logs.
  - Respond to support tickets and manage platform-level settings.
- Registration includes name, email, and password, with email verification planned in a future milestone.

Prioritized High-Level Functional Requirements

Priority 1

1. User

1.1. A user shall be able to create an account.
1.2. A user shall be able to securely log in to their account.
1.3. A user shall be able to recover or reset their password.
1.7. A user shall be associated with one or more accounts.
1.10. A user shall be able to view and edit their financial data history.

2. Account

2.1. An account shall store income entries for a user.
2.2. An account shall store expense entries for a user.
2.3. An account shall belong to one and only one user.
2.4. An account shall track a balance value.
2.7. An account shall have a transaction history log.

3. Bank

3.1. A bank shall be linked to one or more user accounts.
3.2. A bank shall provide transaction data to the system.

4. Transaction

4.1. A transaction shall be classified as income or expense.
4.2. A transaction shall belong to exactly one account.
4.3. A transaction may be manually entered or synced from a bank.
4.4. A transaction shall be categorizable by the user or AI.

5. Subscription

5.1. A subscription shall be tied to a recurring transaction.

6. Budget

6.1. A budget shall be created for one or more spending categories.
6.2. A budget shall belong to one and only one account.
6.3. A budget shall allow setting and tracking of goals.

7. Task

7.1. A task shall be assigned to a user account.

7.2. A task shall be marked completed upon user action.

8. Reward System

8.1. A user shall be able to earn rewards for completing tasks.

8.2. A user shall be able to redeem rewards through the system.

10. Notification System

10.1. The system shall send reminders for logging receipts.

10.2. The system shall deliver daily, weekly, or monthly financial summaries.

11. Administrator

11.1. An administrator shall be able to suspend or maintain site operations.

11.2. An administrator shall be able to view flagged transactions.

Priority 2

1. User

1.4. A user shall be able to select between manual or bank-linked financial tracking.

1.8. A user shall be classified as a customer, premium, tester, or administrator.

2. Account

2.6. An account shall be able to categorize transactions.

2.9. An account shall track subscriptions and bill payments.

2.10. An account shall be able to forecast end-of-month balances.

3. Bank

3.3. A bank shall store balance and credit data for users.

4. Transaction

4.5. A transaction shall be editable and deletable by the user.

4.6. A transaction shall be linked to one or more receipts.

5. Subscription

5.2. A subscription shall be viewable in a centralized dashboard.

5.3. A subscription shall trigger periodic spending alerts.

6. Budget

6.4. A budget shall support notifications based on spending limits.

6.5. A budget shall be able to generate monthly recaps.

7. Task

7.3. A task shall be of one type: savings, investing, setup, admin, testing.

8. Reward System

8.3. A user shall be assigned a level based on completed actions.

10. Notification System

10.3. The system shall notify users of upcoming bills or subscriptions.

10.4. The system shall notify users of overspending events.

10.5. The system shall alert users when savings goals are off track.

11. Administrator

11.3. An administrator shall be able to respond to support tickets.

11.4. An administrator shall be able to access user account data.

Priority 3

1. User

1.5. A user shall be able to customize their interface (e.g., dark mode).

1.6. A user shall be able to receive AI-generated financial guidance.

1.9. A user shall be able to interact with the AI chatbot for financial assistance.

2. Account

2.5. An account may be connected to one or more banks.

2.8. An account shall support spending limits and alerts.

4. Transaction

4.7. A transaction may be associated with a reimbursement request.

7. Task

7.4. A recommended task shall be generated based on user activity or AI analysis.

8. Reward System

8.4. A user shall be able to view rankings if competitive mode is enabled.

9. AI Assistant (LemonAid)

9.1. The AI assistant shall analyze user transactions and spending habits.

9.2. The AI assistant shall simulate different financial scenarios for planning.

9.3. The AI assistant shall forecast recurring charges and balances.

9.4. The AI assistant shall recommend saving opportunities and budget changes.

9.5. The AI assistant shall provide suggestions for uncategorized transactions.

11. Administrator

11.5. An administrator shall be able to manage and moderate platform use.

Mockups/Storyboards


M2Mockup-01.jpg


M2Mockup-02.jpg


M2Mockup-03.jpg


M2Mockup-04.jpg


M2Mockup-05.jpg


M2Mockup-06.jpg


M2Mockup-07.jpg

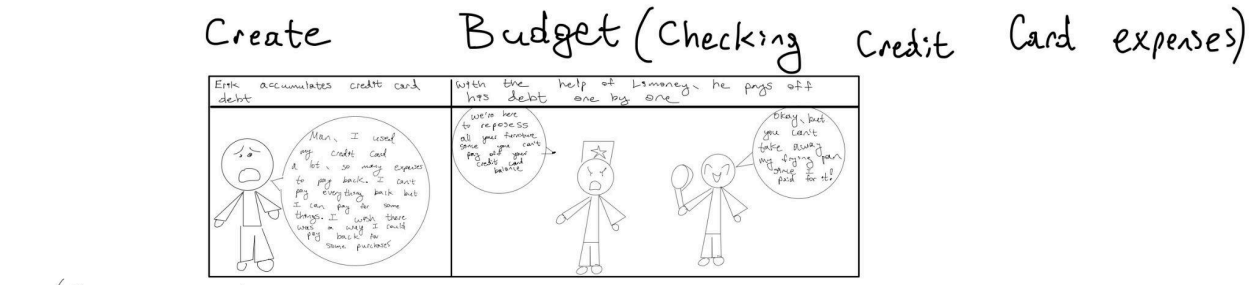

M2Mockup-08.jpg


M2Mockup-09.jpg

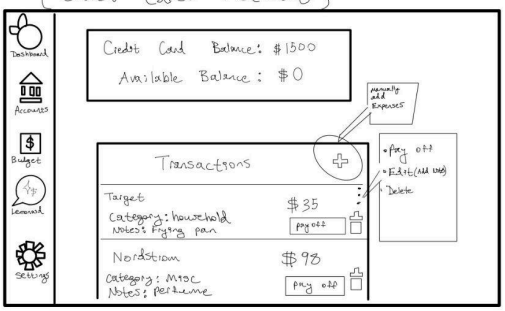

M2Mockup-10.jpg


M2Mockup-11.jpg

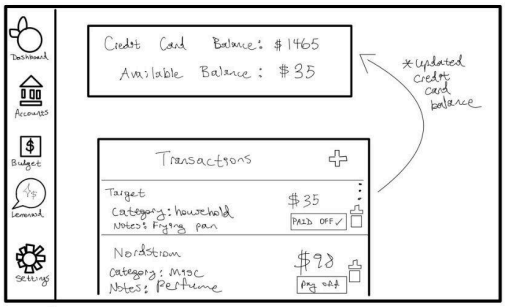
Mockups/Storyboards



(Credit Card Account)

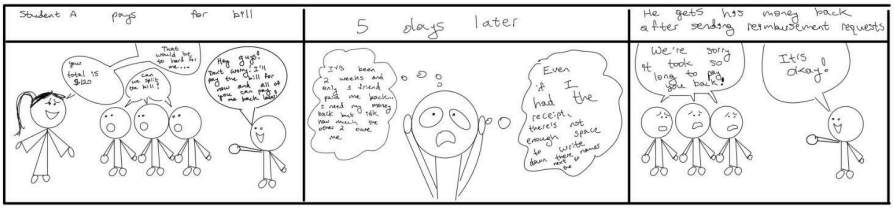


1. Erik goes to a specific Credit Card Transaction. The pay off buttons on each transaction indicate which expenses haven't been paid off

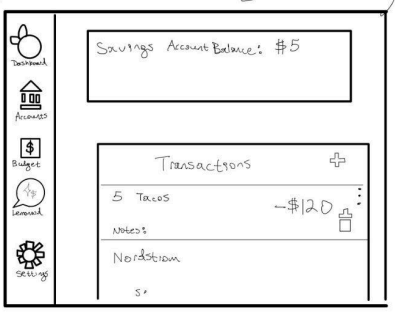


2. Next, Erik pays off one expense, so the transaction gets marked as "paid off". Then the credit card balance is updated

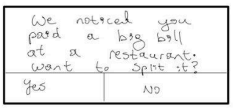
Reimbursement Requests



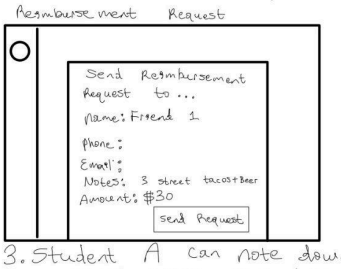
(Student A's Savings Account



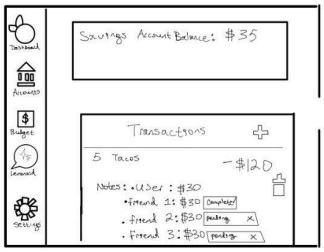
1. Student A opens up the transaction for the restaurant bill



2. He also receives a notification for large bills over $100. He is prompted to split the bill



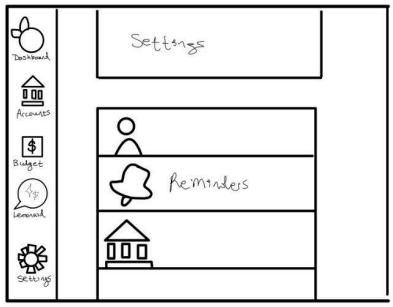
3. Student A can note down how much money each his of friends owe him and send them reimbursement requests



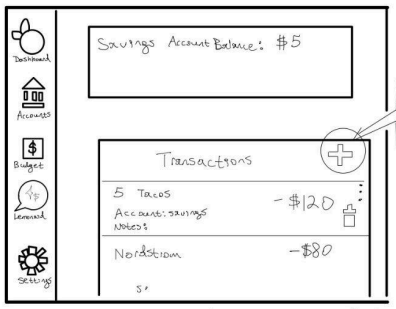
3. When the friend(s) pay(s) back Student A, Student A can manually mark the request as completed and the account balance will be updated. Incomplete reimbursement requests will be marked as "pending".

Log Transactions

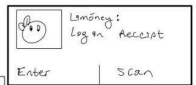
Account Section



or



Reminder for web app



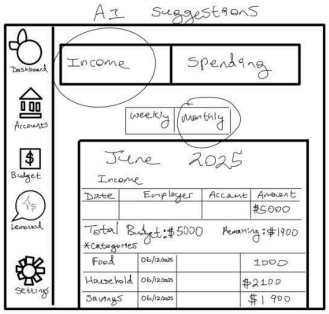
2. These are the reminders to log in receipts sent to Erik exclusively from Limoney. The scan feature will use a camera, scan photo of Erik's receipt, and auto fill the data fields such as

1. Erik can set reminders via settings or the account transaction dashboard

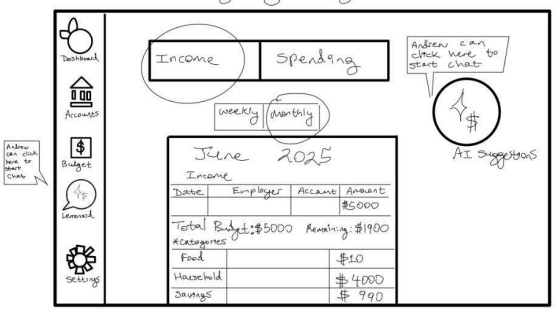
Security Concern

Modified Budgeting Page

AI Suggestions



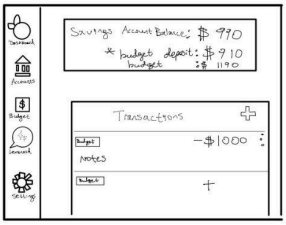
Budgeting page



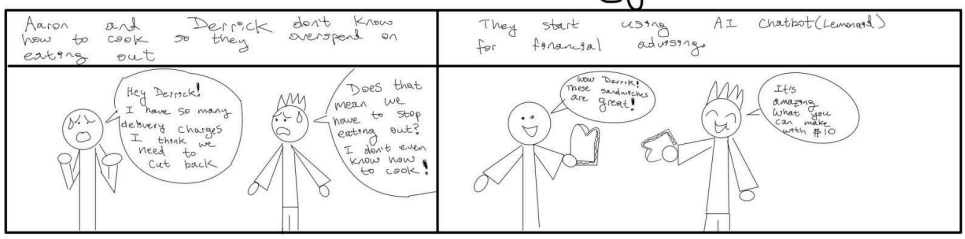
1. Andrew opens budget page where he can manually enter his monthly or weekly income in the income section

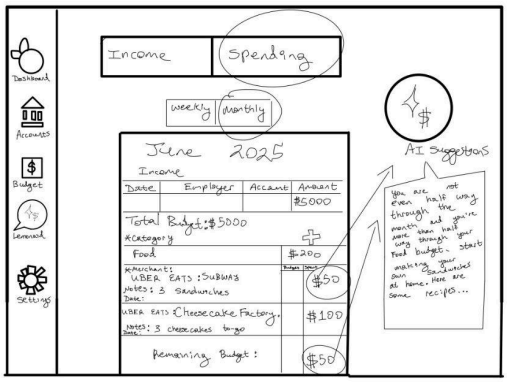
2. Andrew, being a premium user, can access the Lemonaid chatbot through the budgeting page or nav bar

3.



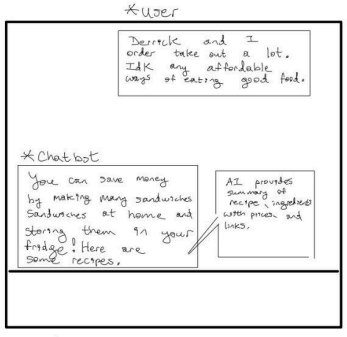
Receive AI suggestions



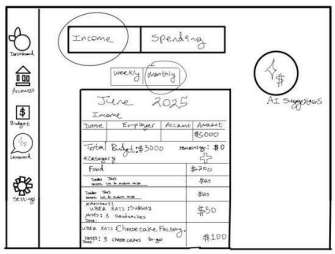


1. This is the "Spending" section of the budget, subtracting the budget total from the income section with transactions from accounts. The AI warns how he's overspending and the visual page helps him realize

*User



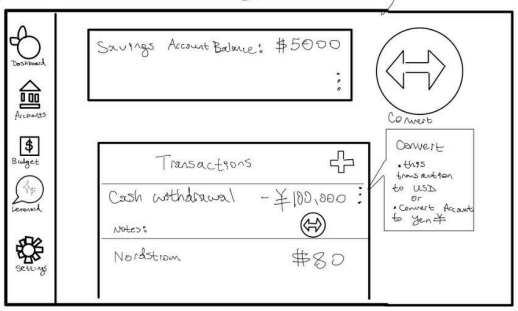
2. Aaron consults with the Lemonaid chatbot, asking how he and Derrick can stick to their budget.



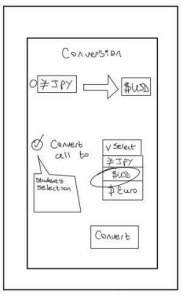
3. The AI chatbot updates the income section of the budget to help Aaron plan his food expenses for the rest of the month.

Currency Exchange

(Student A's Savings Account)

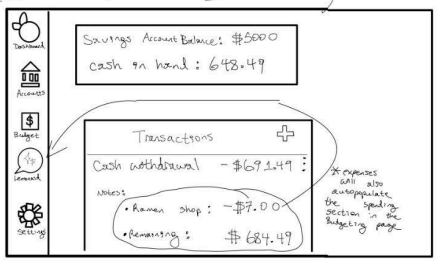


1. American Study Abroad student manually logs in the ATM cash withdrawal in ¥yen. He has three buttons to help him make currency conversions: the arrow button on the transaction itself (usually triggered only if transaction is in foreign currency), the big "convert" button on the right, and the drop down menu.



2. The student has the option of converting all transactions/expenses into $USD or converting just that one expense into $USD.

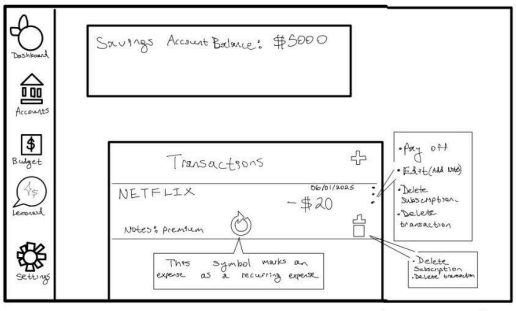
(Student A's Savings Account)



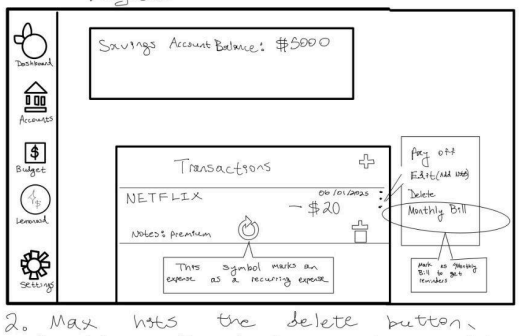
3. The student converts that cash withdrawal into $USD and can log in how he spent that cash which will also autopopulate his budgeting page.

Tracking Ongoing Expenses

Regular User

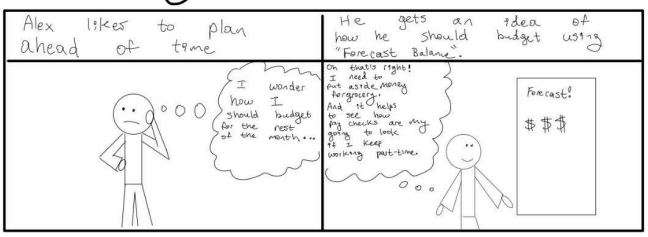


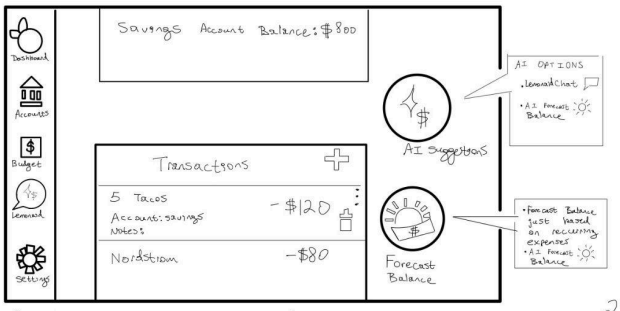
1. Max sees the subscription (recurring expense) on his account marked by a curvy symbol. Max is a premium user, so the Lemoney web app can detect subscriptions through his linked bank accounts, but if he was a regular user, he would have manually mark his subscription as a recurring expense



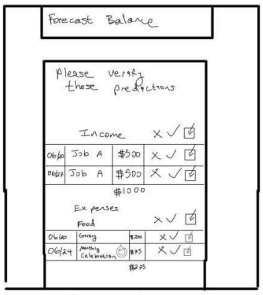
2. Max hits the delete button, directing him to the merchants platform to delete his subscription since the Lemoney webapp. He also had the option to put reminders for his subscriptions

Predicting Balances End of Month



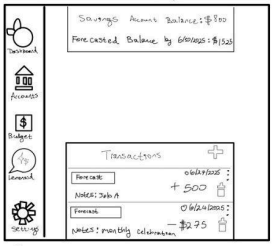


1. Alex goes to his Savings account. His balance sits at $800 at the beginning of the month but he wants to know what his balance will look like at the end of the month given his upcoming expenses and paychecks (income)



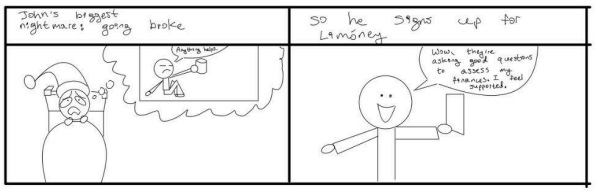
2. The forecast presents Alex options to accept, reject, or edit for his bank balances and budget.

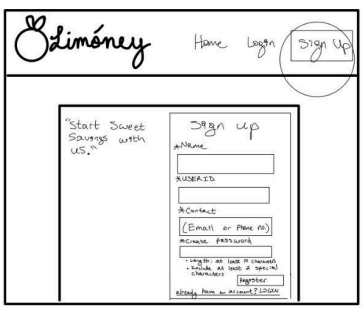
Modified Budgeting page and Accounts pages



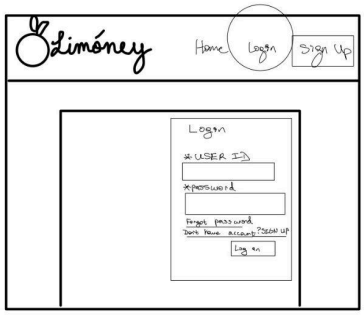
3. Alex's bank balance is not updated, rather the upcoming expenses are displayed as "forecast" and there's a calculated balance for the end of the month seperate from the current balance.

Signing up / Intro UI

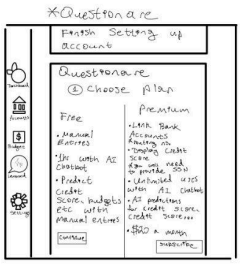




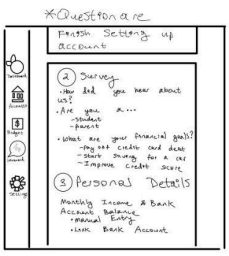
1. John goes to Sign up page.



or 1. John went to sign up account and was already told he has an account so he logs in.

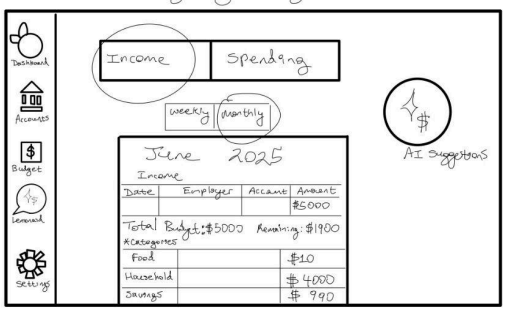


2. John chooses to be a premium user and answers the questionair so the Limoney webapp can give him a personalized experience based on his



Chatbot / AI assistance (Lemonaid)

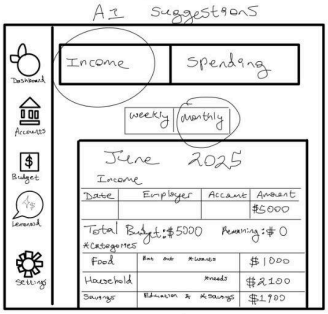
Budgeting page



1. John opens his budget to start planning for June 2025. Overwhelmed, he clicks on "AI suggestions" which directs him to the Lemonaid Chatbot. This demonstrates accessibility: a reminder for John that he seek support when feeling overwhelmed

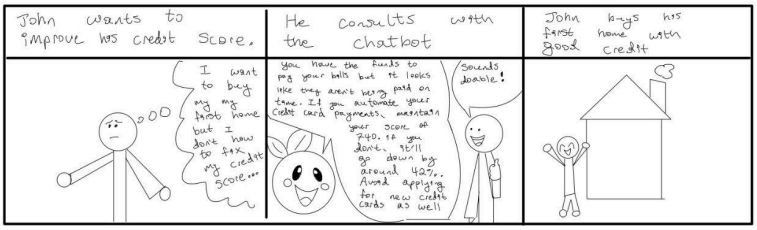
2. He asks the Chatbot for advice and it gives a response based on John's Account Balances, recurring expenses, income, demographic, credit card debt, etc.

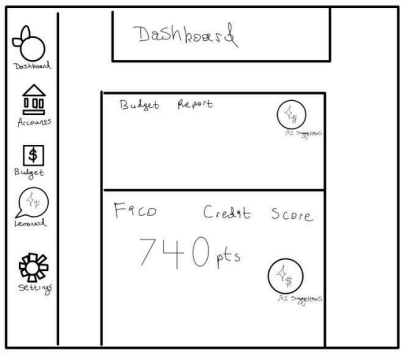
Modified Budgeting page AI Suggestions



3. The AI updates John's budget within their conversation. based upon agreements

Credit Score Inquiry





1. John can check his Credit score on the dashboard using his

2. John consults with Lemonaid about improving his Credit score. Although Lemonaid provides suggestions, John will be responsible for automating his credit card bills.

High-Level System Design

High Level Database Architecture

Initial Database Requirements:

1. User

```
1.1.A User shall be associated with 0 to many Accounts.

1.2.A User shall be able to create and manage 0 to many Tasks.

1.3.A User shall be rewarded with 0 to many Rewards.

1.4.A User shall receive 0 to many Notifications.

1.5.A User shall be able to submit 0 to many SupportTickets.

1.6.A User shall be assigned 0 to many Levels.

1.7.A User shall have 0 to many LemonAidLogs.

1.8.A User can be a subtype Administrator.

1.9.A User may receive responses from an Administrator via associated
SupportTickets.
```

2. Account

```
2.1.An Account shall belong to exactly 1 User.

2.2.An Account shall be associated with 0 to many Transactions.

2.3.An Account shall be linked to 0 to many Budgets.

2.4.An Account shall be linked to 0 to many Subscriptions.

2.5.An Account shall be associated with 0 to many Banks through an
AccountBankLink associative entity.
```

3. Bank

```
3.1.A Bank shall be linked to 0 to many Accounts via AccountBankLink.

3.2.A Bank may serve multiple users through its linked Accounts.
```

4. Transaction

```
4.1.A Transaction shall belong to exactly 1 Account.

4.2.A Transaction shall have 0 or 1 associated Receipt.

4.3.A Transaction shall be associated with 0 or 1 ReimbursementRequest.
```

5. Receipt

```
5.1 A Receipt shall be associated with exactly 1 Transaction.
```

6. ReimbursementRequest

```
6.1.A ReimbursementRequest shall be linked to exactly 1 Transaction.
```

7. Subscription

```
7.1.A Subscription shall belong to exactly 1 Account.
```

8. Budget

```
8.1.A Budget shall belong to exactly 1 Account.
```

9. Task

```
9.1.A Task shall be assigned to exactly 1 User.
```

10. Reward

```
10.1.A Reward shall be associated with exactly 1 User.
```

11. Notification

```
11.1.A Notification shall be sent to exactly 1 User.
```

12. LemonAidLogs

```
12.1.A LemonAidLog shall be created by exactly 1 User.
```

13. SupportTicket

```
13.1.A SupportTicket shall be created by exactly 1 User.

13.2.A SupportTicket may be responded to by 0 or 1 Administrator.
```

14. Administrator (ISA of User)

```
14.1.An Administrator shall inherit its identity from a User (PK = FK).

14.2.An Administrator may manage 0 to many SupportTickets.

14.3.An Administrator may be assigned to 0 to many Users for admin-
level actions.
```

DBMS Selection: We will use **MySQL** because it is a popular and reliable database system that works well with web apps. It can handle our financial data and user information easily, and it's a good fit for the tools we're using like Node.js and Express. Also, it is accepted by the CTO.

Database Organization:

1. **User (Strong)**

- Attributes:
  - userID (PK): INT
  - name: VARCHAR(40)
  - email: VARCAR(40)
  - password: VARCHAR(255)
  - user_type: ENUM(standard,admin)
- Relationships:
  - 1 to 0 Many with **Account**
  - 1 to 0 Many with **Task**
  - 1 to 0 Many with **Reward**
  - 1 to 0 Many with **Notification**
  - 1 to 0 Many with **SupportTicket**
  - 1 to 0 Many with **Level**
  - 1 to 0 Many with **LemonAidLogs**

2. **Account (Strong)**

- Attributes:
  - accountID (PK): INT
  - userID (FK): INT
  - balance: INT
  - account_type: ENUM(checking, savings, credit)
- Relationships:
  - Many to 1 with **User**
  - 1 to 0 Many with **Transaction**
  - 1 to 0 Many with **Budget**
  - 1 to 0 Many with **Subscription**
  - 0 Many to 0 Many with **Bank**

3. **Bank (Strong)**

- Attributes:
  - bankID (PK): INT
  - name: VARCHAR(40)
- Relationships:
  - Many to Many with **Account**

4. **Transaction (Strong)**

- Attributes:
  - transactionID (PK): INT
  - accountID (FK): INT
  - amount: INT
  - type: ENUM(income, expense, transfer)
  - category: VARCHAR(40)
  - date: DATE
- Relationships:
  - 0 Many to 1 with **Account**
  - 1 to Many with **Receipt**
  - 1 to 0,1 with **ReimbursementRequest**

5. **Receipt (Weak)**

- Attributes:
  - receiptID (PK): INT
  - transactionID (FK): INT
  - date_uploaded: DATE
  - image: blob
- Relationships:
  - 0 Many to 1 with **Transaction**

6. **ReimbursementRequest (Weak)**

- Attributes:

    - requestID (PK): INT
    - transactionID (FK): INT
    - status: ENUM(pending, approved, rejected)
- Relationships:
    - 0,1 to 1 with **Transaction**

7. **Subscription (Strong)**

- Attributes:
    - subscriptionID (PK): INT
    - accountID (FK): INT
    - name: VARCHAR(40)
    - amount: INT
    - interval: ENUM(daily, weekly, monthly, yearly)
    - next_due_date: DATE
- Relationships:
    - 0 Many to 1 with **Account**

8. **Budget (Strong)**

- Attributes:
    - budgetID (PK): INT
    - accountID (FK): INT
    - limit_amount: INT
    - start_date: DATE
    - end_date: DATE
- Relationships:
    - 0 Many to 1 with **Account**

9. **Task (Strong)**

- Attributes:
    - taskID (PK): INT

    - userID (FK): INT
    - type: ENUM(daily, weekly, one-timer, custom)
    - status: ENUM(pending, completed, overdue)
- Relationships:
    - 0 Many to 1 with **User**

10. **Reward (Strong)**

- Attributes: rewardID (PK), userID (FK), points
- Relationships:
    - 0 Many to 1 with **User**

11. **Notification (Strong)**

- Attributes:
    - notificationID (PK): INT
    - userID (FK): INT
    - type: ENUM (info, alert, reminder)
    - content: TEXT
    - date: DATETIME
- Relationships:
    - 0 Many to 1 with **User**

12. **LemonAidLogs (Strong)**

- Attributes:
    - logID (PK): INT
    - userID (FK): INT
    - output: TEXT
    - timestamp: TIMESTAMP
    - rating: ENUM(1,2,3,4,5)
- Relationships:
    - 0 Many to 1 with **User**

13. **SupportTicket (Strong)**

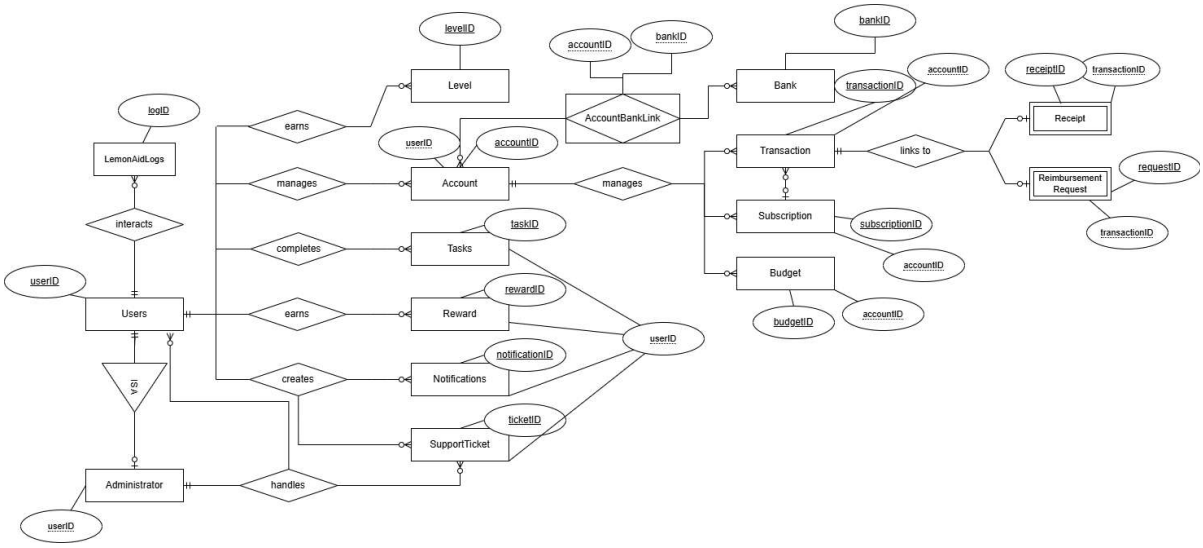
- Attributes:
  - ticketID (PK): INT
  - userID (FK): INT
  - subject: VARCHAR(40)
  - message: TEXT
  - status: ENUM (open, in progress, closed)
  - admin_response: TEXT
- Relationships:
  - 0 Many to 1 with **User**
  - 0 Many to 1 with **Administrator**

14. **Administrator (Strong)**

- Subtype of User
- Attributes:
  - adminID (PK, FK from User): INT
  - permissions: TEXT
- Relationships:
  - 1 to 0 Many with **User**
  - 1 to 0 Many with **SupportTicket**

15. **AccountBankLink (Associative)**

- Attributes: accountID (FK), bankID (FK)



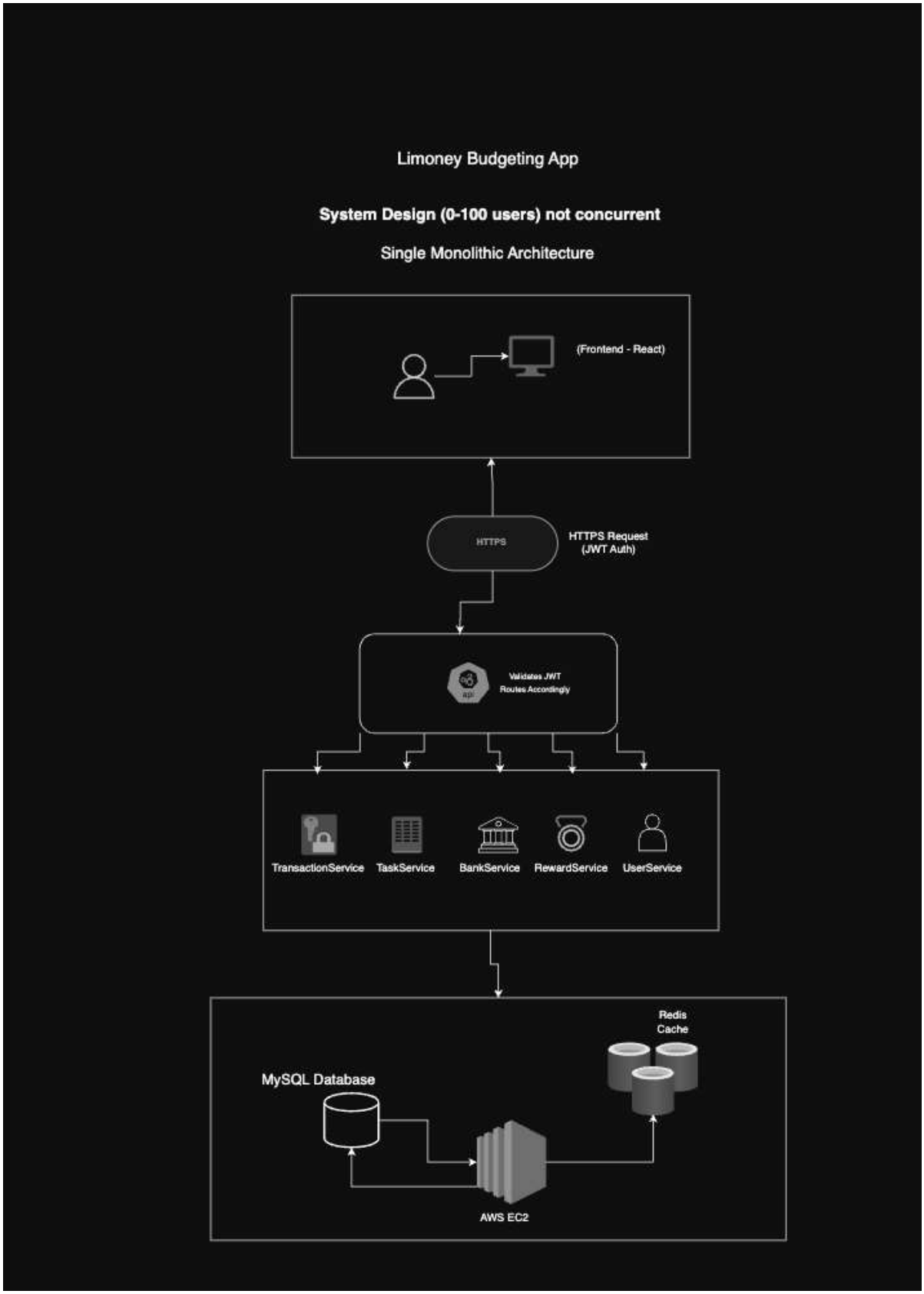
Media Storage

For now, we plan to store images as BLOBs for things like receipts, since they are small and directly tied to transactions. We do not currently need to store video, audio, or GPS data, but if needed in the future, we may switch to a file system to handle larger files more efficiently.

Backend Architecture

SMALL SCALE DESIGN

The Limoney Budgeting App for 0-100 (not concurrent) users uses a single monolithic architecture. The frontend (React) communicates via HTTPS (JWT Auth) to a backend API, which validates JWT and routes requests to services such as TransactionService, TaskService, BankService, RewardService, and UserService. All services interact with a single MySQL database (hosted on AWS EC2) and an optional Redis cache for session/AI caching.



Small Scale Architecture

MEDIUM SCALE DESIGN

For 10-1000 concurrent users, the system introduces microservice replication, load balancing, and read replicas for the database. The backend is horizontally scalable with Dockerized microservices, an auto-scaling NGINX load balancer, and health checks. The MySQL database uses master-read replica architecture for scalability, and Redis is used for caching AI and frequent queries. Improvements include fault tolerance, health checks, and reduced DB pressure.

System Design (10-1000 concurrent users)

Challenges (10–1000 concurrent users):

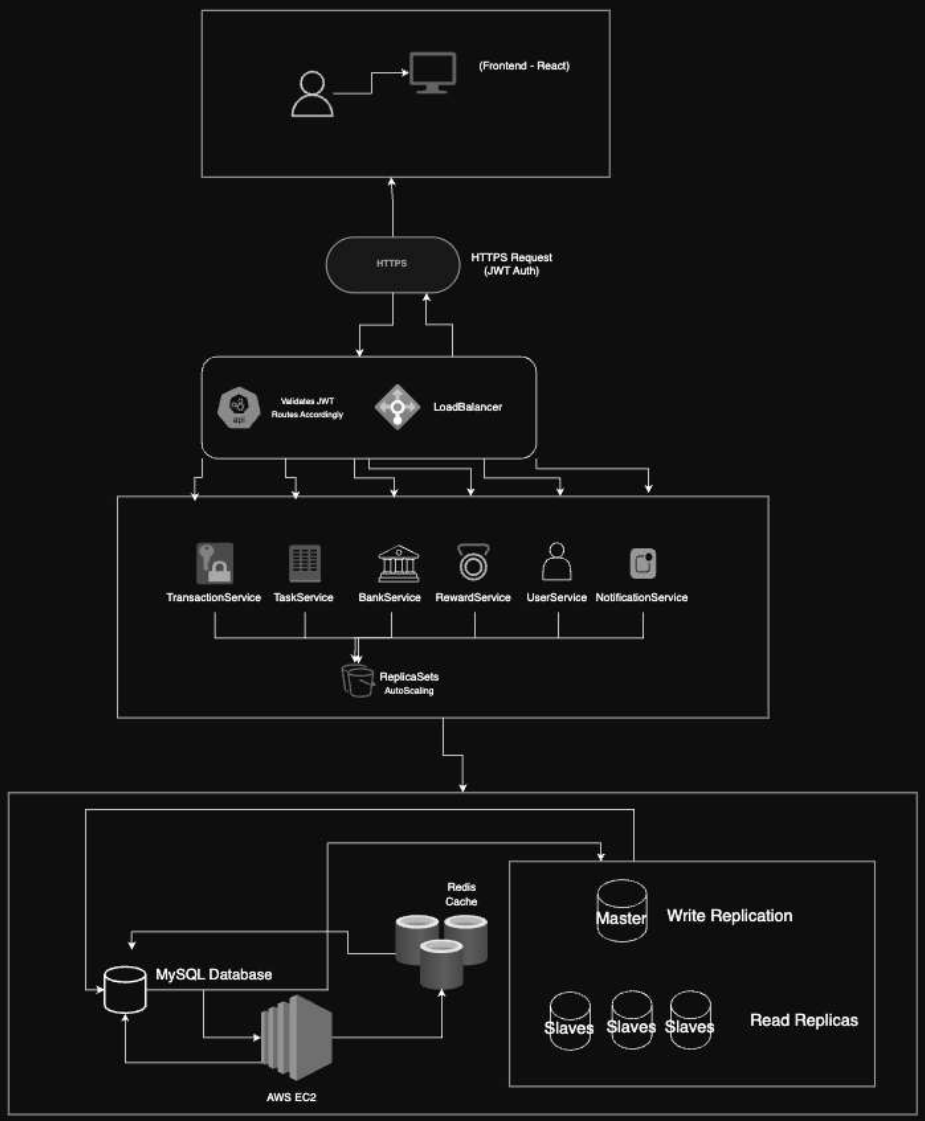
1. **Data Consistency** (Multiple DBs increase risk of partial updates and eventual sync issues.)

2. **Service Coordination** (Complex logic spans services (e.g., completing a task > updating rewards)

3. **Network Latency** (More inter-service traffic = higher request duration)

4. **Management Overhead** (Deploying and monitoring many services requires orchestration, logs, metrics)

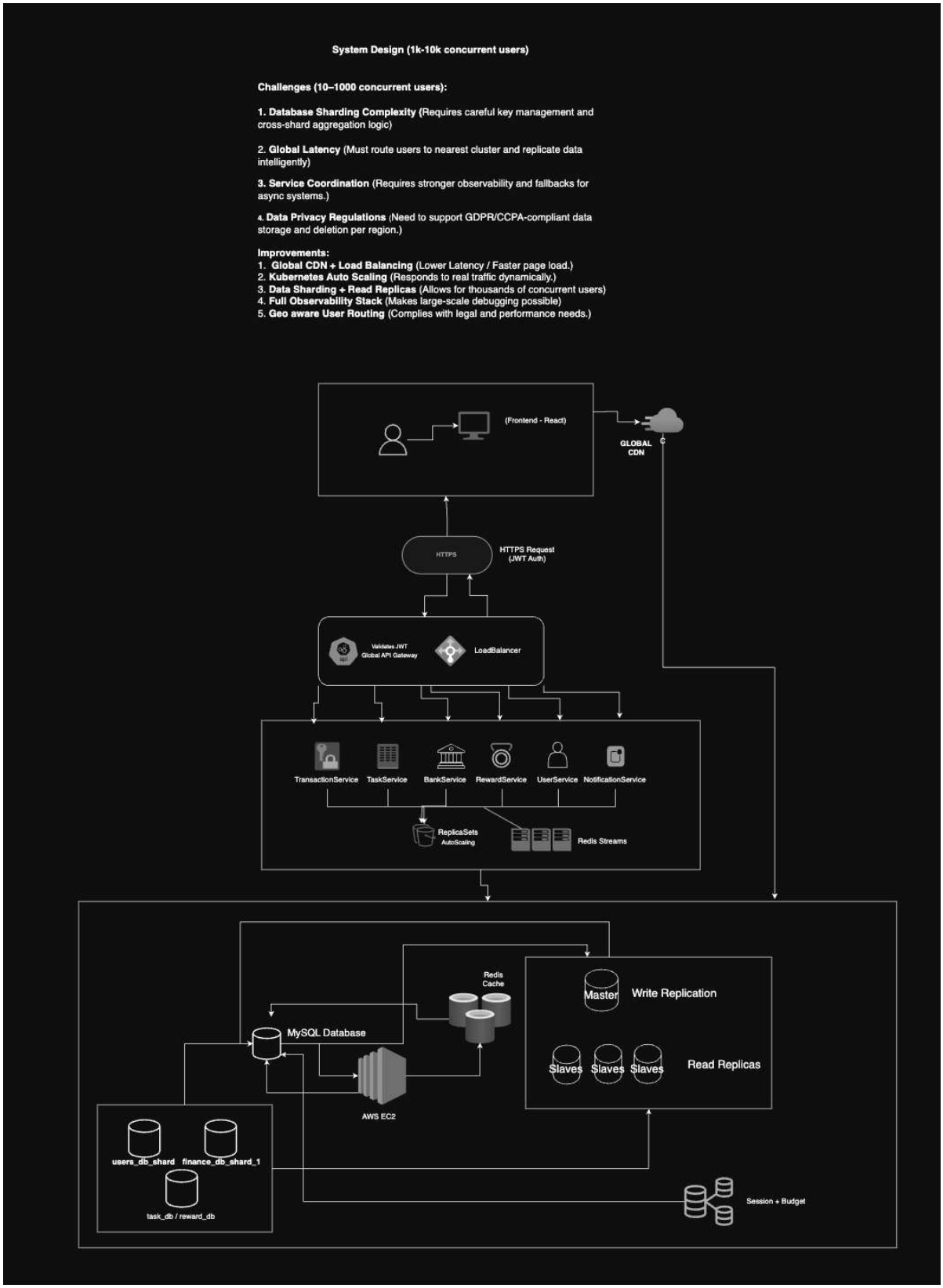
Improvements:
1. **MicroService Replication** (Fault tolerance, zero downtime during container restarts.)
2. **LoadBalancer + HealthCheck** (Keeps system available and responsive.)
3. **Split DB + Read Replicas** (Faster queries, reduced lock contention)
4. **Redis Cache** (Decreased DB pressure, faster response times)



Medium Scale Architecture

LARGE SCALE DESIGN

For 1k-10k concurrent users, the system leverages Kubernetes-managed microservices, global CDN, and database sharding by user region. The backend is orchestrated with Kubernetes for auto-scaling and CI/CD. The database layer uses sharding and a mix of read/write replicas, with multi-layer Redis caching. Security is enhanced with rate-limiting, API key enforcement, and mutual TLS between services. The architecture supports global latency reduction and compliance with data privacy regulations.



Large Scale Architecture

ARCHITECTURE SUMMARY

1. Microservices Architecture Limóney uses a domain-driven microservices architecture where each major business capability is its own independently deployable service: UserService, BankService, BudgetService, SubscriptionService, TaskService, AIRecommendationService, RewardService

Services communicate via REST APIs and asynchronous messaging (for AI or reward triggers)Backend System Design.

2. Load Balancing Using an NGINX reverse proxy or AWS ALB, traffic is routed evenly. Health checks ensure services are available. Load balancing supports horizontal scaling, allowing the system to increase availability and responsivenessBackend System Design.
3. Caching Strategies Cache-aside pattern for database query results (ex, user budget history).

Uses LRU eviction for memory optimizationBackend System Design.

4. Reliability and Fault Tolerance Each service runs in its own container managed by Docker / Kubernetes, allowing independent failure recovery.

Circuit breakers protect the system from cascading failures (Resilience4j).

Use Retry + Timeout policies to stabilize connections.

5. Containers Docker containers isolate each service.

Portable environments between local → staging → production.

Kubernetes handles orchestration, deployment, health checks, scaling, and rolling updatesBackend System Design.

6. Data Replication and Consistency Master-Slave replication for MySQL improves read throughput.

Sensitive operations (ex., TransactionService, SavingsService) use synchronous writes for consistency.

Some services may rely on eventual consistency for performanceBackend System Design.

7. Security Considerations JWT authentication for all user requests.

Role-Based Access Control (RBAC): Admins vs. Customers.

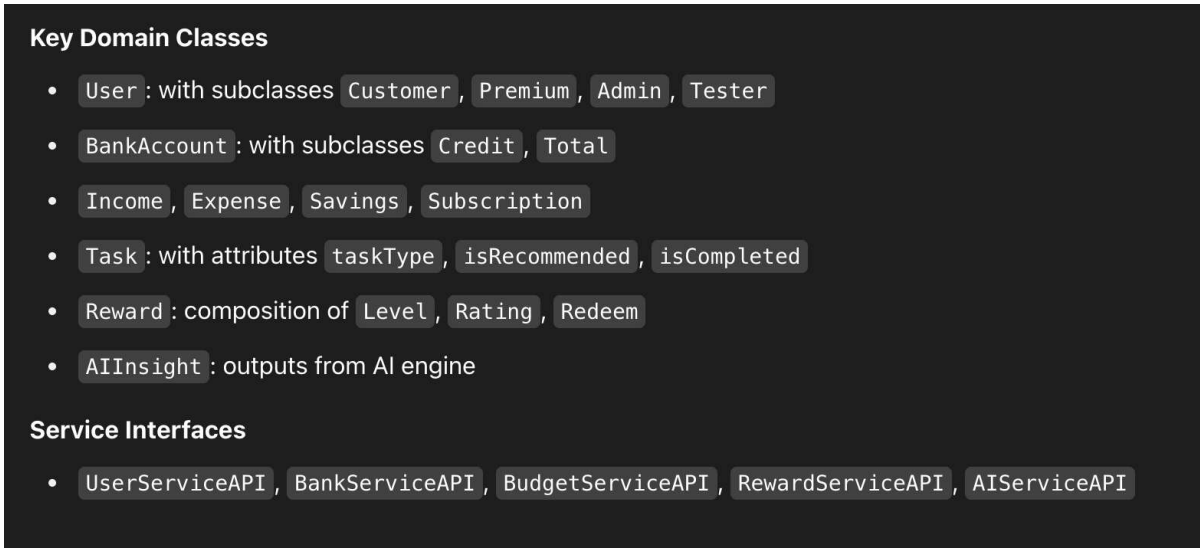
Mutual TLS between services.

Rate limiting + logging at the gateway.

Data encryption (at rest using AES-256, in transit via TLS)Backend System Design.

UML DESIGN

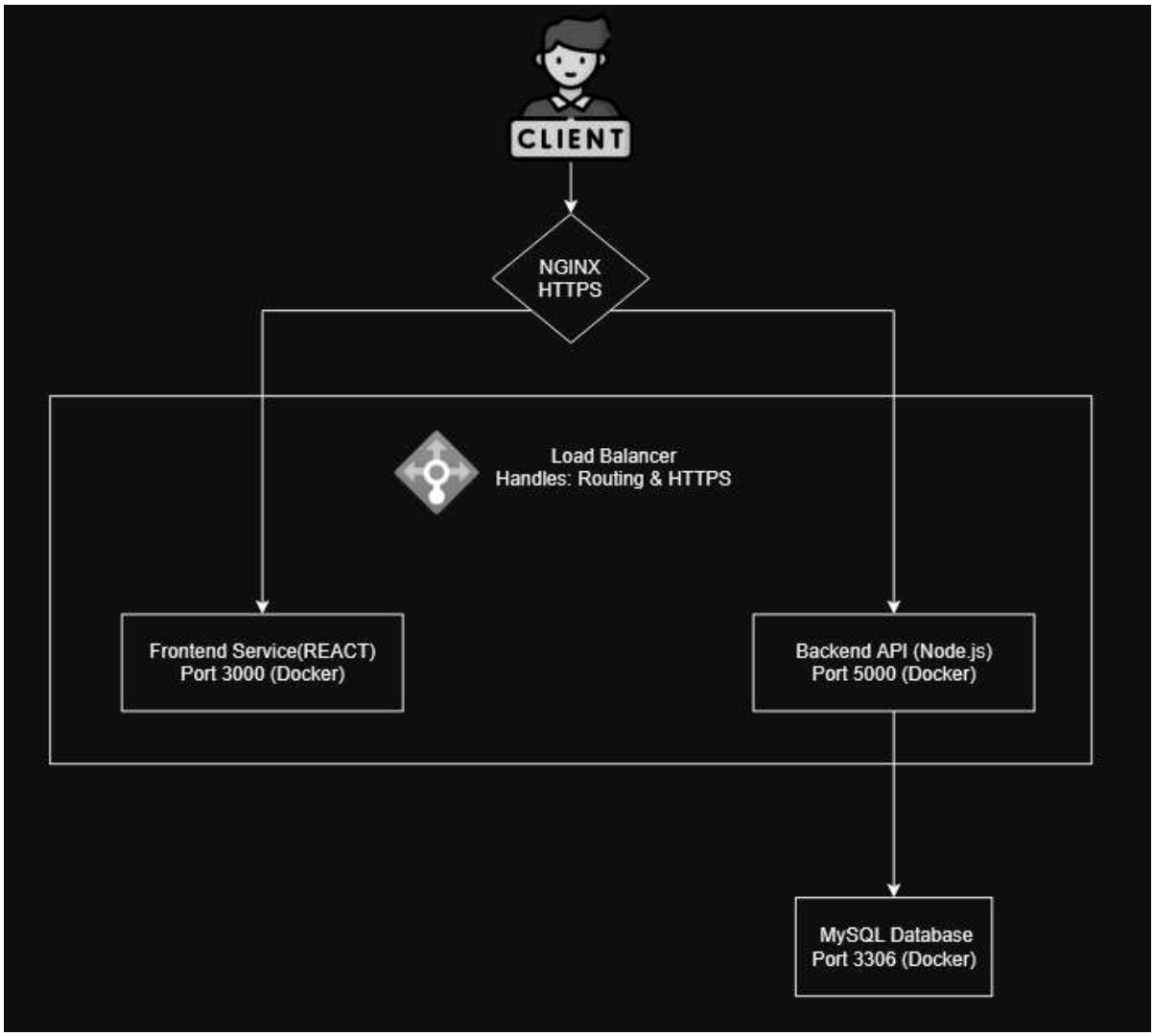
The following UML diagram illustrates the key domain classes and service interfaces for the Limoney backend system. It includes classes such as User (with subclasses), BankAccount, Income, Expense, Savings, Subscription, Task, Reward, and AIInsight, as well as the main service APIs.



UML Design

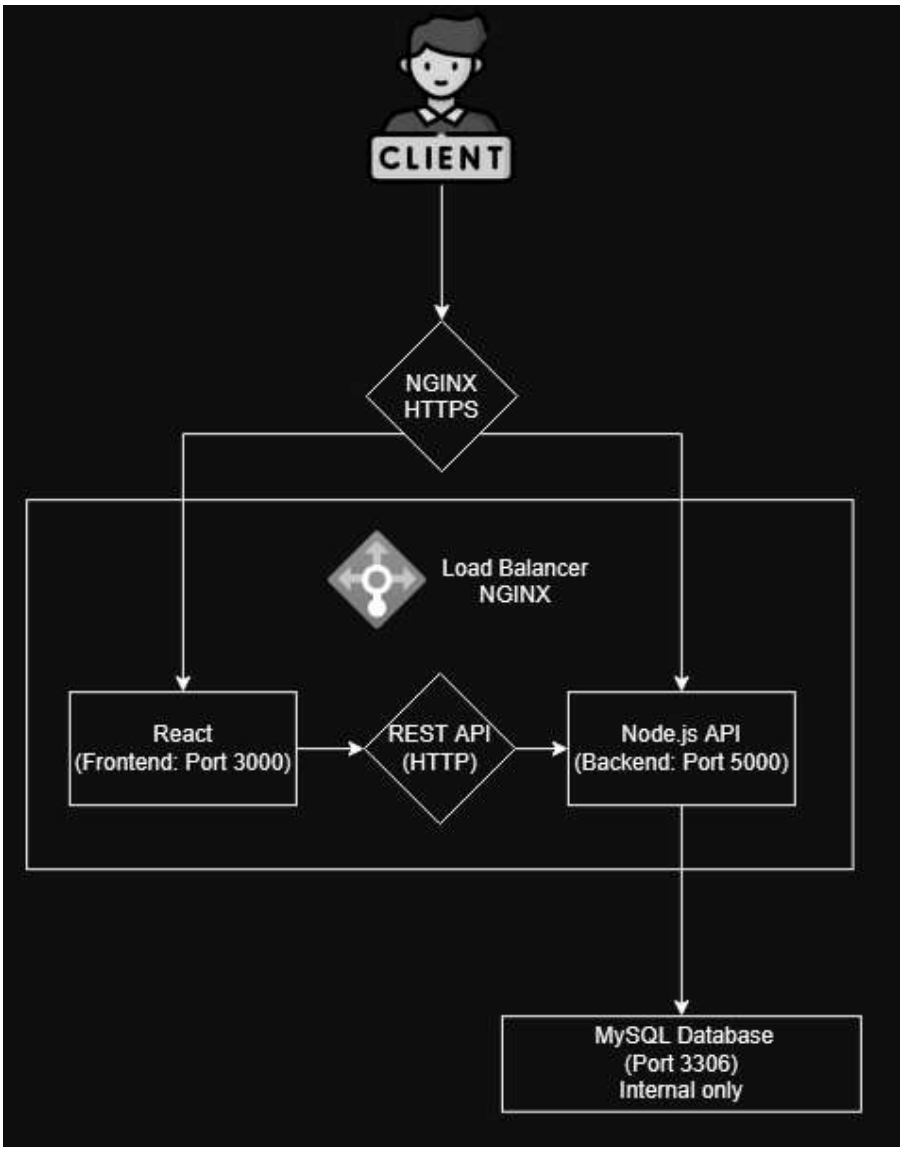
High Level Application Network Protocols and Deployment Design

Network & Development Diagram



This diagram shows how external traffic flows through the system. The client communicates with the server via HTTPS, which is handled by NGINX acting as a reverse proxy. NGINX routes requests to either the React frontend or the Node.js backend API. The backend connects internally to a MySQL database, which is not exposed to the public.

Application Networks Diagram

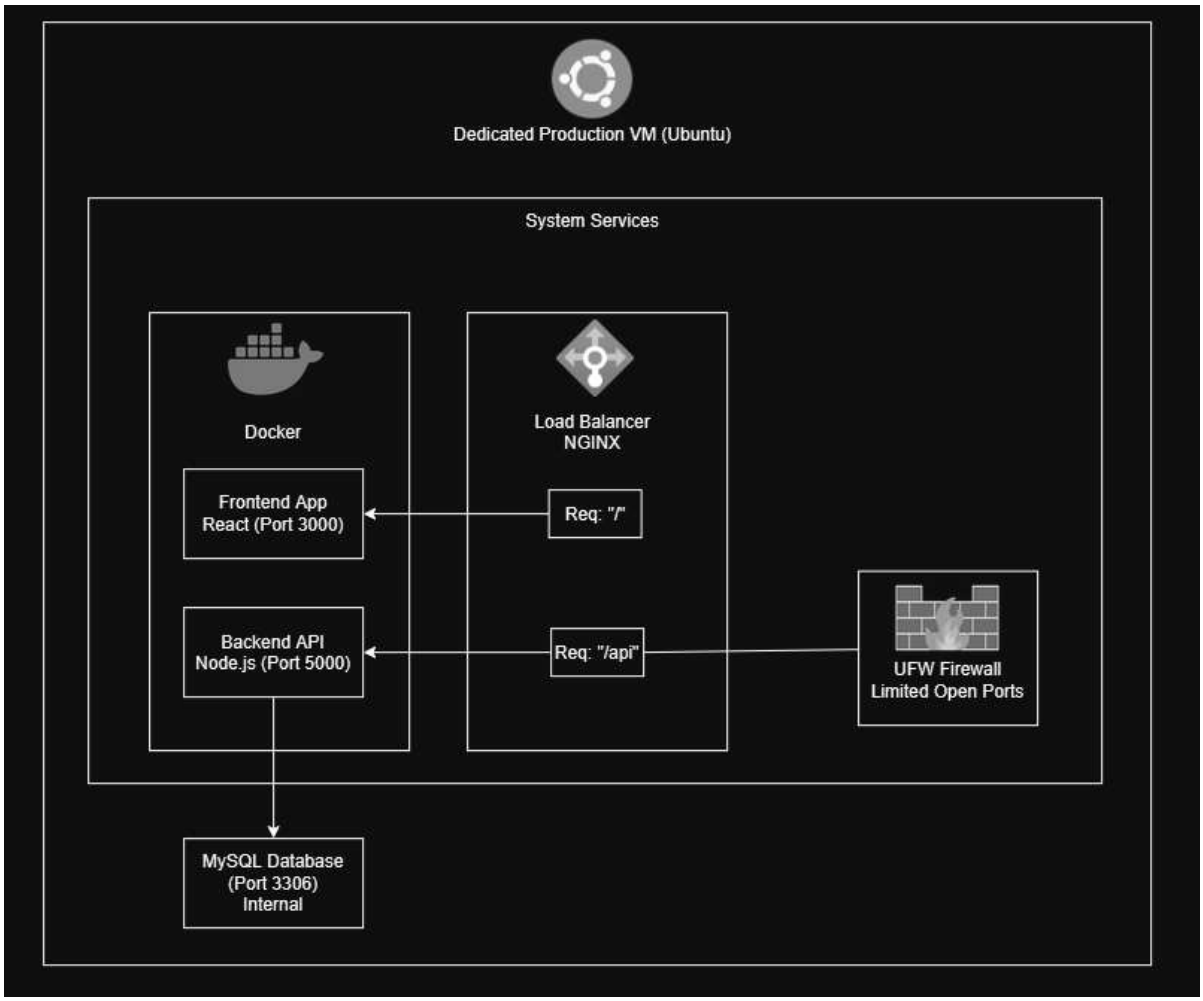


Protocols

Network uses HTTPS for securing client access with NGINX, routing to React frontend and Node.Js backed. The backend accesses MYSQL database internally over TCP/IP. Services run on Docker bridge network, with security enforced through SSL/TLS, CORS. and a UFW firewall that restricts access only to essential

ports to the application to work. NGINX is the main gateway and proxy which isolates the internal services from the public.

Deployment Diagram



This diagram illustrates the system's structure on a dedicated Ubuntu server. Docker manages the React frontend and Node.js backend processes. NGINX handles incoming traffic and routes it to the appropriate service. A UFW firewall restricts network access, allowing only essential ports. The backend securely connects to an internal or cloud-hosted MySQL database.

Integration with External Components

External Components:

- no third party API's so far

Internal Libraries

- bycrypt: for hasing passwords
- dotenv: for ".env" variable handling
- cors: for seucring cross-origin request
- mysql12: for database connection

High Level APIs and Main Algorithms

We will develop several high-level APIs that handle the core functionality of the financial management system:

- **Account & Transaction Management API**: Allows users to securely manage accounts, log income/expenses, categorize transactions, and attach receipts.
- **Budget & Subscription API**: Helps users create, update, and track budgets and recurring payments with due-date reminders.
- **Task & Reward System API**: Supports gamification by letting users complete tasks and earn reward points based on activity.
- **Notification API**: Delivers alerts, reminders, and system messages tied to user activity and financial goals.
- **Support & Admin API**: Enables users to submit support tickets and allows admins to manage and respond to them.
- **LemonAid AI Interaction API**: Logs interactions between users and the LemonAid assistant and allows users to rate each AI response.

Main Algorithms and Processes

- **AI Rating System**: After each interaction with the LemonAid assistant, users can rate the response on a scale of 1–5. These ratings will be used to analyze the quality of AI output over time. We may later implement a feedback loop to highlight poor responses for admin review or AI fine-tuning.
- **Spending Categorization**: Transactions will be auto-labeled into categories. We plan to use simple keyword matching initially, with the option to evolve into a smarter, pattern-based system.
- **Budget Alerts**: Users will be notified when spending approaches a budget limit. This will be based on periodic checks of current spending vs. set limits.
- **Reward Calculation**: Users earn points by completing financial tasks or goals. The system will calculate points based on task type, frequency, and completion streaks.

Software Tools and Frameworks

We are currently using:

- **Node.js + Express** (backend API framework)
- **MySQL** (relational database)
- **React or HTML/CSS/JavaScript** (frontend)
- **Docker** (for deployment)
- **AWS EC2 with Ubuntu 22.04** (cloud hosting)
- **Let's Encrypt + Certbot** (SSL)

We may later integrate:

- **Firebase Authentication** (optional, for user auth and password resets)
- **OpenAI API** (for LemonAid assistant)
- **Tesseract OCR or Google Vision API** (if we add automatic receipt scanning)

Any new tools will be approved by the instructor before implementation.

Key Project Risks

- **Skills risks**
  - Each individual of the group has familiarity with one or more of the concepts of this project from the technical documentation to the prototype itself. What everyone shares is the desire to learn and push oneself. So, if someone comes across a task that they are unfamiliar with, they are likely to reach out several sources to learn.
- **Schedule risks**
  - Time is very limited for all the members given that we all have responsibilities outside of this class. However, with proper planning and communication, we can meet the deadlines.
- **Technical risks**
  - Given that our cloud server host has very limited vCPU and RAM, there will be difficulties. With just our mere about us page in milestone 1, there was an issue with overloading the RAM that it crashed the server. To solve this, we will try to finish prototypes earlier to give the backend team time to adjust for a smooth deployment for this and future milestones.
- **Teamwork risks**
  - If people are not communicative with what they are struggling with, there will be a delay in progress. So, to minimize that from happening, there are consistent check-ins on everyone to invite open communication and to keep everyone up to date.
- **Legal/content risks**
  - To legally use all required software in Limóney, we must ensure any UI kits, AI tools, and APIs are properly licensed, especially for commercial use. Services like OpenAI require a paid plan if used in a production setting. As well, we must include clear disclosures that your AI assistant provides suggestions, not professional financial or legal advice.

Project Management

In Milestone 2, project management started off rough because we had a set up on Notion and then the free trial expired faster than expected. After that, we quickly moved over to Gitbook since we were familiar with it due to the technical documentation. However, the free trial expired and made it excessively inconvenient to use it for project management. From then, the team lead stayed active with making sure everyone was on top of things by messaging them for updates. For milestone 3, we will attempt at using Jira (very similar to Notion), which is another project management tool. If the team lead read the free trial rules correctly, it should be able to last until the summer semester is over.

List of Team Contributions

Name	Contributions	Score (1-10)
Emily Perez	wrote key project risks; wrote title page; finalized high-level functional requirements; organized and finalized technical documentation; assisted with setting up the directory; helped establish connection between frontend, backend, and database	10
Ishaank Zalpuri	wrote data definitions; helped with database architecture; helped with high-level functional requirements	10
Andrew Brockenborough	created table of contents; updated readme.md file in application folder; helped with high level apis and main algorithms; helped with high-level functional requirements	10
Dani Luna	created todo list; wrote data definitions; helped with database architecture; created mockups/storyboards; worked on the search bar algorithm for m2c2; helped with high-level functional requirements	10
Jonathan Gonzalez	helped with high-level functional requirements; worked on backend architecture	10

Gene Orias	establish connection between frontend, backend, and database; completed the signup aspect of m2c2; updated credentials files; encrypted server data into code (.env); worked on high-level application network protocols and deployment design; helped with high-level functional requirements	10