Introduction and overview

Problem Statement:

- Many people find it difficult to manage their personal finances because they lack clear visibility and personalized, actionable advice. Manually tracking every expense is tedious, and most finance apps offer generic guidance that rarely leads to lasting behavioral change. As a result, users often remain unaware of wasteful spending habits, miss opportunities to save money, and feel stressed about their financial future.

Proposed Solution:

- An AI-powered financial coach that analyzes a user's raw transaction data and turns it into simple, personalized insights. The goal is to help users understand their spending, make smarter financial decisions, and feel more in control of their money.

Tech Stack

Frontend Technologies:

- Next.js - React framework for creating beautiful web apps
- Typescript - statically typed superset of Javascript
- TailwindCSS - CSS library with a lot of utilities
- Recharts - chart visualization library)

Backend Technologies:

- FastAPI - Python framework for web APIs
- Python 3 - versatile, OOP programming language
- Uvicorn (ASGI server)

AI/ML Technologies:

- Pandas - Python library for data analysis + manipulation
- Numpy - Python library for numerical computations
- Facebook Prophet - Open Source framework for time series forecasting
- Google Gemini API - LLM for AI powered insights and natural language processing

Data design

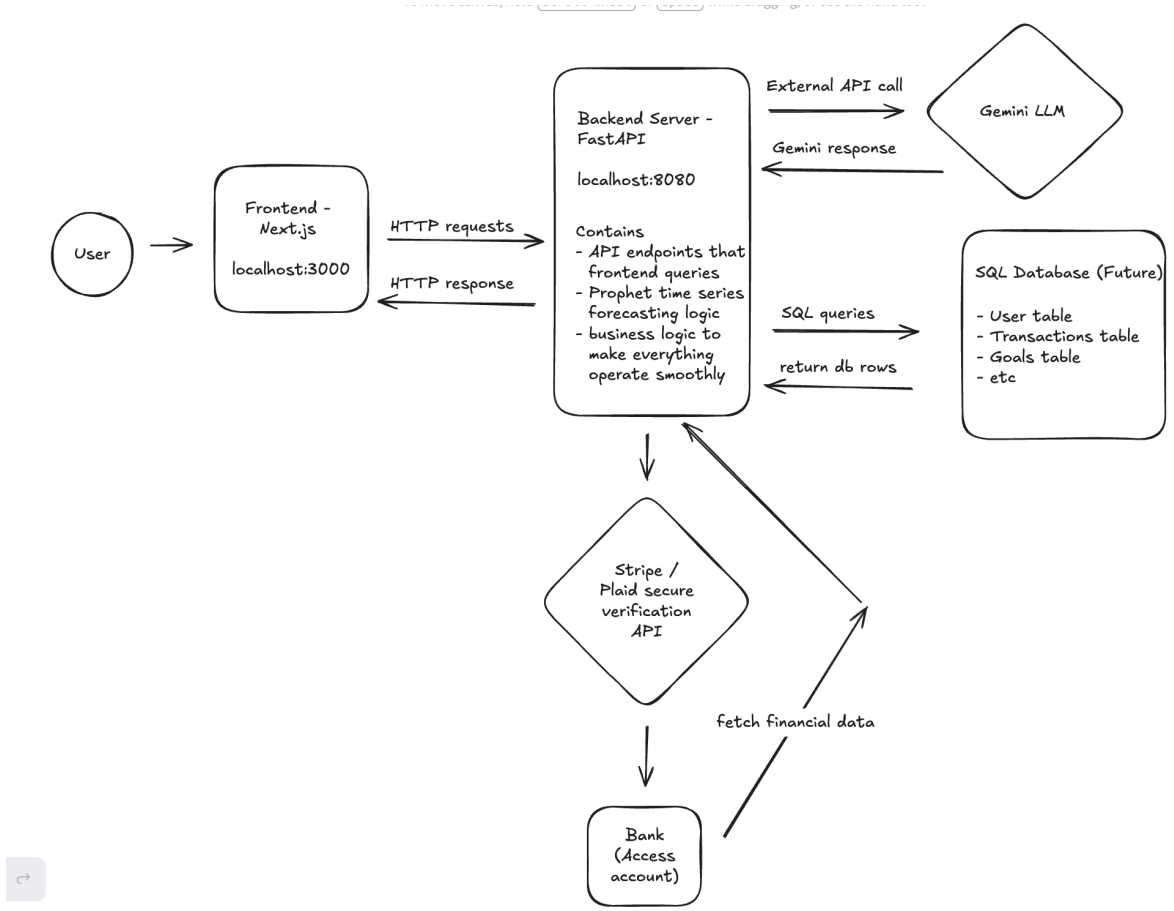
Due to the short nature of this hackathon, I decided to take a simpler approach to data design and direct most of my focus to feature development instead.

I used a CSV file to store transaction data. I used OpenAI's ChatGPT to generate 1 year's worth of synthetic real-world transaction data. This data included a purchase timestamp, purchase amount, merchant name, spending category, payment channel, and pending flag.

I also used JSON-serializable in-memory storage for user savings goals. Goals are stored in a Python dictionary structure with the schema {user_id: {goal_id: Goal}}, allowing for CRUD operations without database overhead. Each goal includes fields for goal name, target amount,
deadline, current savings, priority level, monthly income, and income type.

System Architecture

(This is an optimized architecture diagram, including some features I WANT to implement so in an ideal world my architecture would look like this).



Features

Main Feature Set:

1. Intelligent Spending Insights:
    a. Description
        i. Analyzes transaction history to surface spending patterns and trends, delivering personalized insights with specific dollar amounts and actionable advice (e.g., "You've spent $120 on coffee this month — brewing at home could save you over $1,000 a year!").
    b. Implementation
        i. Pandas aggregates spending statistics and detects notable patterns like spending spikes or category dominance. Gemini 2.5 Flash converts these findings into friendly, conversational insights.
2. Personalized Goal Forecasting:
    a. Description
        i. Predicts whether you'll reach your savings goal by the deadline based on historical spending patterns and existing goals. Also shows expected outcomes with confidence intervals on an interactive graph. Suggests specific categories to cut if you're off track.
    b. Implementation
        i. Facebook Prophet forecasts future spending using time series analysis. Savings are derived by subtracting predicted spending from user-provided income.
3. Subscription Tracker:
    a. Description
        i. Identifies recurring subscriptions from transaction history, flags "gray charges" (small forgotten subscriptions), detects price increases, and highlights potential trial-to-paid conversions.
    b. Implementation
        i. Pandas groups transactions by merchant and analyzes charge timing and amount consistency to detect recurring patterns matching standard billing frequencies.
4. Investment Capacity Predictor:
    a. Description
        i. Calculates how much you can invest monthly after accounting for taxes, spending, and goal contributions. Provides educational guidance on beginner-friendly options like high-yield savings accounts, index funds, and Roth IRAs.
    b. Implementation

        i. Applies estimated tax rate to gross income, subtracts average spending and goal commitments, and returns investable surplus with plain-language explanations of low-risk investment vehicles.
5. Chatbot:
    a. Description
        i. Natural language interface for querying transactions with questions like "How much did I spend on Uber last month?" Handles typos, maintains conversation context, and responds with specific numbers and comparisons.
    b. Implementation
        i. Gemini 2.5 Flash with function calling interprets queries and executes Pandas operations on transaction data. RapidFuzz handles fuzzy matching for misspelled or mistyped merchant names and categories.

Tradeoffs

1. Function Calling vs RAG for chatbot:
    a. I implemented the chatbot using LLM function calling rather than retrieval augmented generation. Financial queries demand exact numbers which map directly to structured database operations. RAG excels at semantic search over unstructured documents but would return approximate results where precision matters.
2. Facebook Prophet vs linear projection for goal forecasting:
    a. I chose Facebook Prophet for goal forecasting over simpler linear projection. Prophet provides confidence intervals and seasonality detection giving users a range of outcomes rather than a single guess. This adds credible ML to the project and handles spending variations more realistically at the cost of added complexity.
3. In Memory Data Store vs database:
    a. I chose in-memory storage using Python dictionaries and Pandas Dataframes instead of a persistent database. This eliminated setup time and let me focus on features rather than infrastructure. Data doesn't persist across sessions and multi-user support isn't possible but these limitations are acceptable for a demo where judges evaluate features over scalability.
4. Next.js vs React or other frontend frameworks:

a. I chose Next.js over plain React or other frameworks like Vue. Next.js provides a structured project setup with built-in routing and API routes and integrates seamlessly with the React ecosystem. This let me scaffold the frontend quickly without configuring webpack or routing from scratch. The tradeoff is a heavier framework with features I didn't fully use like server-side rendering and static generation. For a single page dashboard app plain React would have been lighter but Next.js developer experience and conventions saved time during a tight hackathon timeline.

Current Limitations

Lack of Authentication:

- Due to this being a hackathon and proof of concept, I decided not to use authentication. In addition, I was not using any real data so authentication would be kind of overkill. Authentication would involve some combination of Plaid/Stripe API with 2 factor authentication (email or phone #) to make sure access to bank account information is thoroughly secure. I have acknowledged this and if given more time to work on the project, it will definitely be a priority.

Lack of Database:

- Since this was a proof of concept and I was testing the features for a single user, I did not opt to use a database. Instead, I used an in-memory data store (using Python dictionary data structure) to maintain state across various transactions. I have acknowledged this and if given more time to work on the project, it will definitely be a priority.

Lack of Security:

- Input is not really sanitized thoroughly. User inputs (chat queries, goal names, income amounts) aren't rigorously validated or sanitized, leaving potential for injection attacks or malformed data (Eg. LLM injection attack). Once again, since this project was more about testing features and less about worrying about malicious attackers, I chose not to spend time on this. However, I have acknowledged this and if given more time to work on the project, it will definitely be a priority.

Challenges

Challenges I faced:

- Initially, I tried using Plaid's Sandbox API in hopes that it would provide realistic transactions
    - However, the transactions were really limited in nature and I wanted more diversity in transaction types + amounts. When I was experimenting with these things, I realized Plaid's free subscription plan would not give me these things.
    - This is why I decided to use ChatGPT to generate a realistic transaction ledger with variance and consumer-like spending habits. This was free and took ~30 seconds
- Prompt engineering the LLM to generate actionable insights
    - Initially, the LLM analyzed user spending habits purely objectively, providing data-driven observations without context or personalization.
    - To make these insights more valuable, I refined the prompts to encourage more subjective, personalized recommendations that account for individual user circumstances and goals
- Initially, I designed the system to process goals sequentially based on creation order. When a user added a new goal (Goal B) while maintaining an existing goal (Goal A), the system would deduct Goal A's allocation before calculating available funds for Goal B.
    - However, I quickly realized this approach had a critical flaw: the arbitrary order in which users created their goals would determine funding precedence, which didn't align with their actual priorities.
    - For example, a user might create a "vacation fund" goal first, then later add a more critical "emergency fund" goal, but the system would prioritize the vacation fund simply because it was created earlier.
    - To address this, I implemented a goal priority system that allows users to explicitly rank their financial objectives. This ensures that when budget constraints arise, the most important goals receive funding first, regardless of creation order.

Future Enhancements

1. Real Bank Integration + Authentication
    a. I want to allow users to securely connect to their bank of choice and access their real transaction data. This would be done through either Stripe or Plaid's secure authentication flow for maximum data security.

2. Persistence
    a. I want to add a persistence layer using a database so the user can save their insights, goals, and other relevant data across sessions. I think a SQL database would make the most sense because the data model has structured, clear relationships.
3. Anomaly Detection
    a. I want to add a feature which flags unusual spending patterns / unrecognised merchant names. I've personally run into situations where fraudulent transactions slipped past me because I didn’t recognize them until much later. A notification system that sends real-time email or text alerts when suspicious activity occurs would help users catch potential fraud early and stay on top of their finances
4. Net Worth Tracking
    a. I would allow users to aggregate their assets (accounts, investments, property) and liabilities. There would be interactive historical net worth trends and projection charts for easy visualization. This would allow users to track their net worth over time and build for a better future by staying more informed.