



Design Documentation



Github Repo: <https://github.com/rimjhimsingh/smart-financial-coach>
Youtube Video Link: https://youtu.be/FMw6KohO_78
Drive Video Link: https://drive.google.com/file/d/138fV0QVotNX8acObLOpsg_XtT42XrNlo/view?usp=sharing

1. Executive Summary

Smart Financial Coach is a full-stack financial intelligence platform designed to transform fragmented transaction data into actionable behavioral insights. Built as a decoupled **React** frontend and **Flask** backend, the application ingests and normalizes disparate financial records (Amex, SoFi, Chase) into a unified memory store for real-time analysis.

Core Capabilities:

- **Unified Executive Dashboard:** Aggregates multi-source data to visualize spend velocity, category mix, and month-over-month deltas using a custom analytics engine.
- **Algorithmic Subscription Audit:** Utilizes heuristic analysis to detect recurring billing patterns, flagging "gray charges" (e.g., trial-to-paid conversions) and calculating annualized costs.
- **Context-Aware AI Copilot:** Features a persistent LLM-backed chat widget that ingests transaction context to answer natural language queries and generate detailed insight cards.
- **Privacy-First Architecture:** Operates on an in-memory data store with localized processing, ensuring high performance without persistent storage of sensitive PII.

Current Prototype Scope:

The MVP demonstrates a complete end-to-end flow: seeding data from raw CSVs, normalizing schemas, identifying anomalies via statistical thresholds, and projecting savings opportunities against user-defined goals.

2. Problem statement and goals

Problem statement

People often know they should budget, but the hard part is turning messy transaction history into clear, actionable decisions. Even when users export transactions, they still face problems:

- **Low visibility:** It is difficult to quickly understand where money is going, what categories are driving spend, and how cashflow changes over time.
- **Hidden waste:** Recurring charges and subscription creep are easy to miss, especially when they blend into normal activity or start as low cost trials.
- **Unclear risk signals:** Large charges can be legitimate, but users need fast, explainable flags for transactions that are unusual for a merchant or reappear after long gaps.
- **Insight fatigue:** Raw charts are not enough. Users benefit from short explanations and concrete next actions, and they need the AI experience to remain useful even when the model is unavailable or rate limited.

This prototype addresses these issues by loading a known transaction dataset into an in memory store and producing a dashboard, recurring charge detections, anomaly flags, and an AI copilot that generates insight cards and answers questions grounded in recent transactions.

Goals

- **One click visibility into finances**

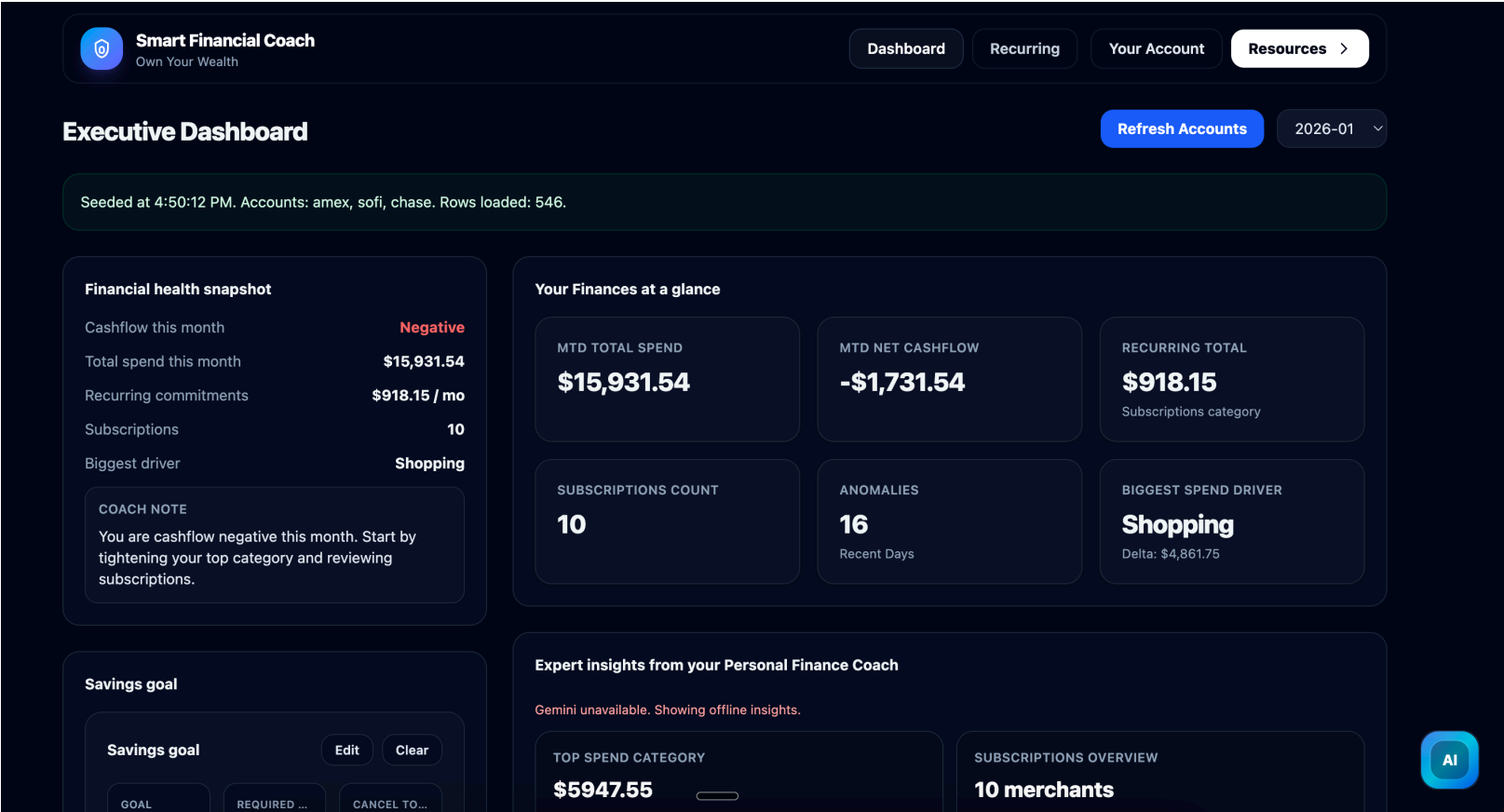
- Provide an executive summary view with month to date spend, net cashflow, recurring total, subscriptions count, anomalies count, and biggest spend driver.
- Provide charts that explain spending and cashflow patterns using spend by category (selected month), money in vs money out by month, and daily spend trend (selected month).
- **Explain what changed and why**
 - Highlight the top month over month category increases (when prior month exists) and identify the top contributing merchants for each increase.
 - Support drilldowns from high level categories into top merchants and top transactions for that category and month.
- **Detect recurring spend and gray charges**
 - Identify recurring merchants using cadence inference (weekly, biweekly, monthly, annual) based on the median gap between charges.
 - Compute subscription metrics that are directly useful for action: average amount, last charged date, occurrences, annualized cost, and a confidence score.
 - Flag likely subscription risks present in the dataset: trial to paid patterns and strict price increase signals for stable series.
- **Flag unusual large outgoing charges with clear reasons**
 - Focus anomalies on outgoing charges only.
 - Restrict candidates to large charges within a recent window.
 - Provide an explanation per anomaly, such as first time charge for a merchant, a long gap since last charge, or a spike versus typical merchant history.
- **Provide an AI copilot that remains reliable**
 - Offer AI generated insight cards for a selected month with a stable contract: always return 5 cards with title, metric, why, action, and drilldown.
 - Provide a chat endpoint that returns strict JSON (answer, bullets, followups) grounded in recent transactions.
 - Degrade gracefully when AI is unavailable using cached insights or deterministic fallback cards generated from the dataset.

Engineering goals

- **Deterministic data foundation**
 - Normalize transactions into a canonical schema (transaction_id, posted_date, merchant, amount, currency, category, account_id, direction).
 - Use an in memory store as a single source of truth for routes and services during a demo run.
- **Robust APIs and error handling**
 - Expose purpose built endpoints for summary, charts, category breakdown, monthly deltas, anomalies, subscriptions, and copilot.
 - Clamp query parameters to safe ranges to prevent oversized responses or invalid inputs.
 - Return consistent JSON responses, including safe parsing of model output to protect the UI from malformed AI responses.
- **Demo friendly behavior**
 - Treat the latest transaction date in the dataset as “today” to make month to date KPIs meaningful without relying on the real current date.
 - Support a seed endpoint that loads the fixed demo CSV files into the running process so the UI can be exercised without external integrations.

3. Feature set selected for the prototype

This prototype focuses on turning a small set of multi account transactions into a usable end to end coaching experience: analytics dashboard, recurring spend detection, anomaly surfacing, and an AI copilot that can summarize and explain insights from the loaded dataset.

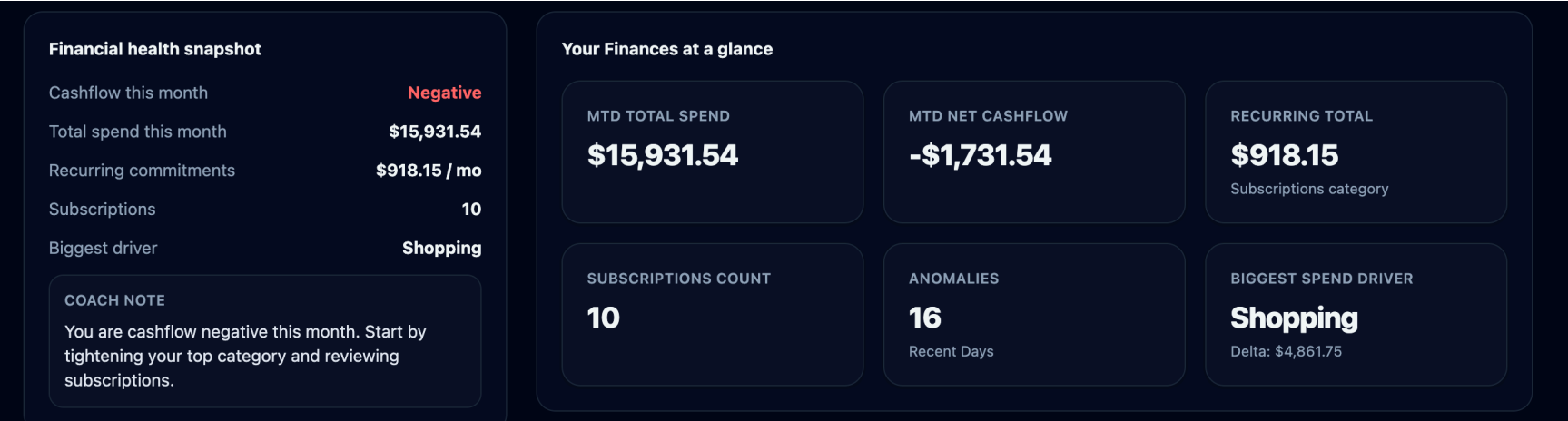


3.1 Multi account demo data ingestion (3 accounts)

- **What the user can do**
 - Load transactions into the app with a single action (Load Demo Data / Refresh Accounts).
 - View dataset status via a lightweight stats call.
- **What the system actually does**
 - Seeds transactions from three local CSVs into an in memory store under account IDs: `amex` , `sofi` , `chase` .
 - Normalizes each CSV into a common schema (`transaction_id`, `posted_date`, `merchant`, `amount`, `currency`, `category`, `account_id`), merges, de duplicates, and stores the merged dataframe for all downstream features.
- **APIs used**
 - `POST /api/seed`
 - `GET /api/stats`
 - `GET /api/transactions?account_id=&limit=`

3.2 Executive dashboard KPIs (month to date)

- **What the user sees**
 - KPI cards that summarize the current month to date, based on the latest date present in the dataset (treated as “today” for demo friendliness).
- **KPIs computed**
 - **MTD Total Spend**: sum of negative amounts in the current month, displayed as positive.
 - **MTD Net Cashflow**: sum of all amounts in the current month (income minus expenses).
 - **Recurring Total**: total spend where `category == "Subscriptions"` in the current month.
 - **Subscriptions Count**: unique subscription merchants across the full dataset where `category == "Subscriptions"` and amount is negative.
 - **Anomalies (last 30 days)**: count of transactions flagged by the anomaly rules.
 - **Biggest Spend Driver**: category with the largest expense delta vs the previous month (expenses only).
- **API used**
 - `GET /api/dashboard/summary`



3.3 Spending trends dashboard (charts + month selection)

- **What the user can do**
 - Select a month from the available months in the dataset.
 - See category spend breakdown for the selected month.
 - See money in vs money out by month across the dataset.
 - See daily spend trend for the selected month (expenses only).
- **Charts returned by backend**
 - `spend_by_category_month`: expenses grouped by category for the selected month.
 - `in_vs_out_month`: money_in, money_out, net for each month in the dataset.
 - `daily_spend_trend`: daily expense totals within the selected month.
 - `available_months`: derived from transaction dates.
- **API used**
 - `GET /api/dashboard/charts?month=YYYY-MM`

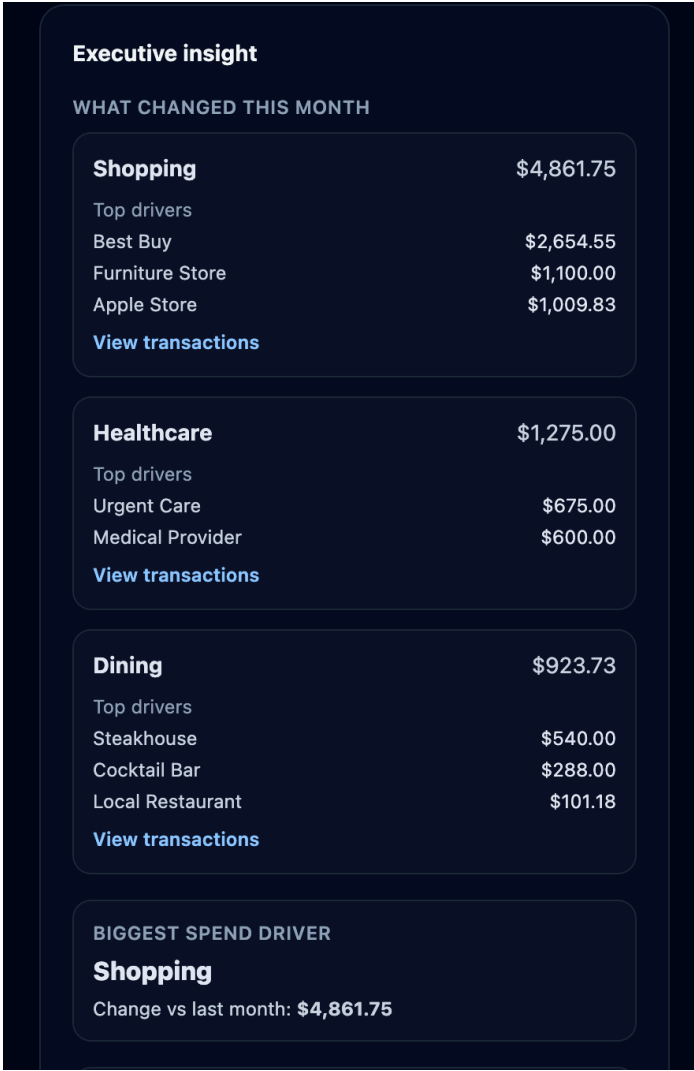


3.4 Category drilldown (from bar chart selection and AI cards)

- **What the user can do**
 - Click a category to drill into: top merchants and top transactions for that category in the selected month.
- **Backend output**
 - Top merchants by total expense (descending).
 - Top transactions by absolute amount (descending), including posted_date and account_id for context.
- **API used**
 - `GET /api/dashboard/category-breakdown?month=YYYY-MM&category=...&merchantLimit=10&txLimit=10`

3.5 Month over month “What changed” insight

- **What the user can do**
 - View the top spending increases by category compared to the previous month, including which merchants contributed most.
- **Backend output**
 - For each top category increase: current total, previous total, delta, and top merchants with their deltas.
- **API used**
 - `GET /api/dashboard/insights/monthly-deltas?month=YYYY-MM&topK=3&merchantsPerCategory=5`



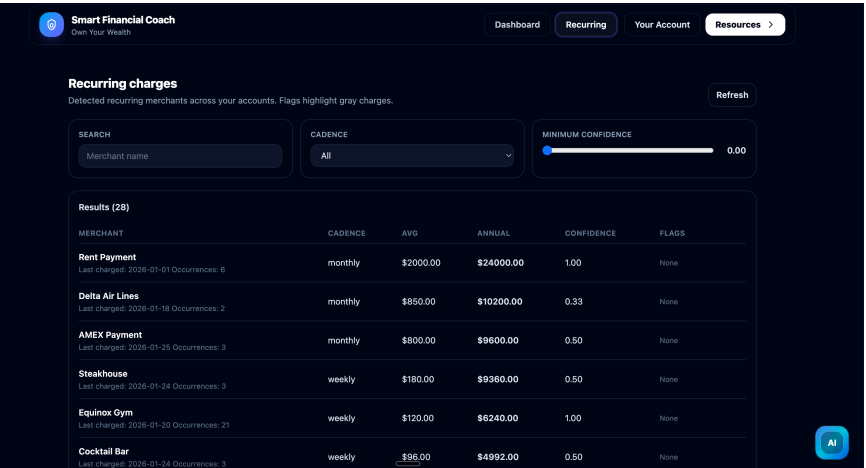
3.6 Anomaly surfacing (rule based)

- **What the user can do**
 - See a short list of anomalies from the last N days (default 30), with a reason string explaining why each was flagged.
- **Rules implemented (outgoing charges only)**
 - Only expenses (`amount < 0`) are considered.
 - Only within the last `days` .
 - Only large charges: `abs(amount) > 500` .
 - Flag if any of these are true:
 - First ever outgoing charge for that merchant.
 - Merchant reappeared after a gap greater than 30 days since the previous outgoing charge.
 - Price spike vs merchant history when enough history exists (at least 3 past points) and the new charge is at least 2x the historical median and at least \$200 higher.
- **API used**
 - `GET /api/dashboard/anomalies?days=30&limit=10`

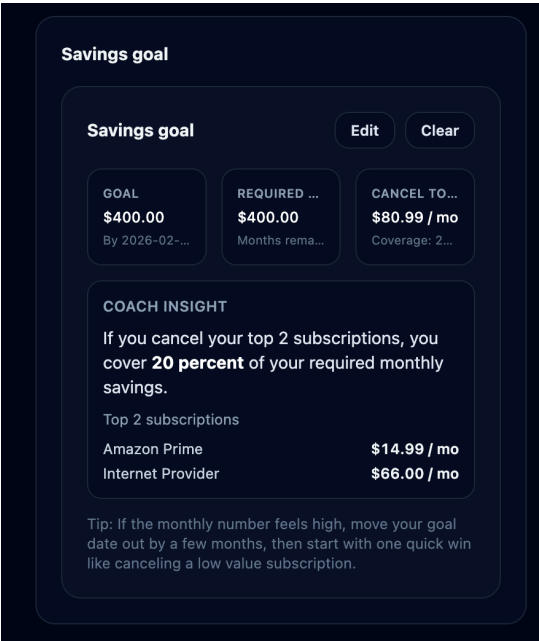
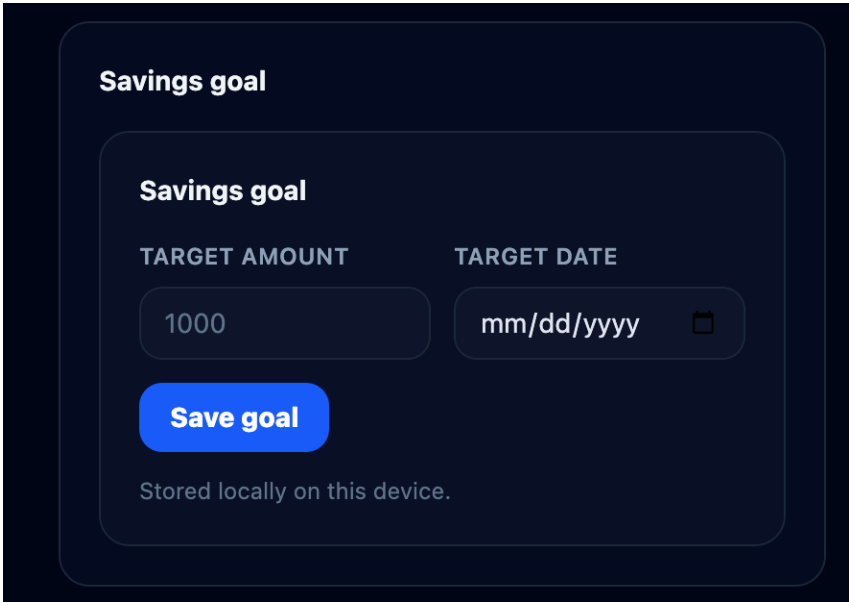


3.7 Subscription and gray charge detector (recurring by merchant)

- **What the user can do**
 - View detected recurring merchants across all loaded accounts.
 - Filter by merchant name, cadence, and minimum confidence.
 - See flags that highlight gray charges.
- **Detection behavior**
 - Groups expenses by merchant, computes day gaps, and matches cadence via median gap with tolerances:
 - weekly, biweekly, monthly, annual
 - Computes: avg charge, annualized cost, last charged date, occurrences, confidence.
 - Flags:
 - **trial_to_paid**: first charge is near zero then later charges are materially higher.
 - **price_increase**: strict detection on stable monthly or annual series.
 - Results are sorted by annualized cost descending to prioritize cancel candidates.
- **API used**
 - `GET /api/subscriptions?min_occurrences=2`



3.8 Savings goal helper (lightweight coaching)



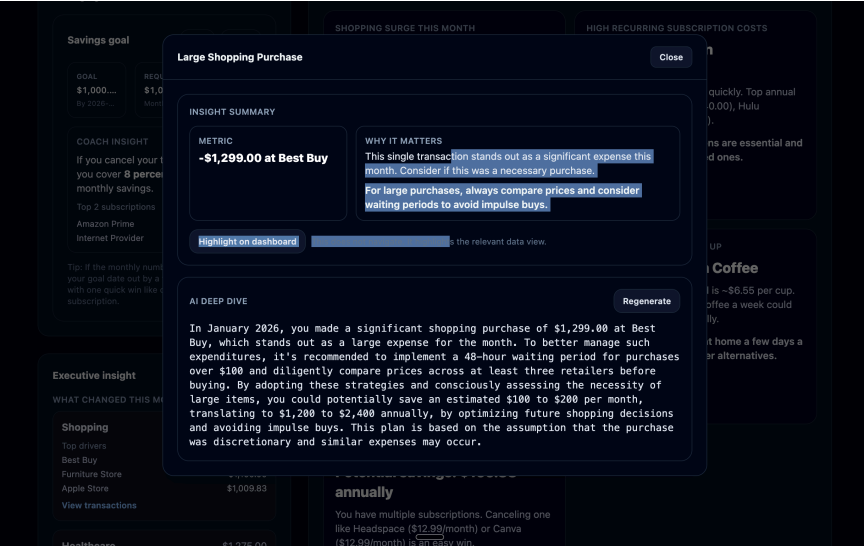
- **What the user can do**

- Enter a savings goal amount and target date in the dashboard sidebar.
 - See a computed monthly savings requirement.
 - Get a suggestion of which recurring charges to cancel first, based on the detected subscription list.
- **How suggestions are chosen**
 - Uses the subscriptions detection output.
 - Filters to subscription like merchants (monthly or annual cadence), a bounded dollar range, and a minimum confidence threshold.
 - Prioritizes by annualized cost and proposes the top two candidates as quick wins toward the monthly goal gap.

3.9 AI insight cards (monthly) with drilldowns

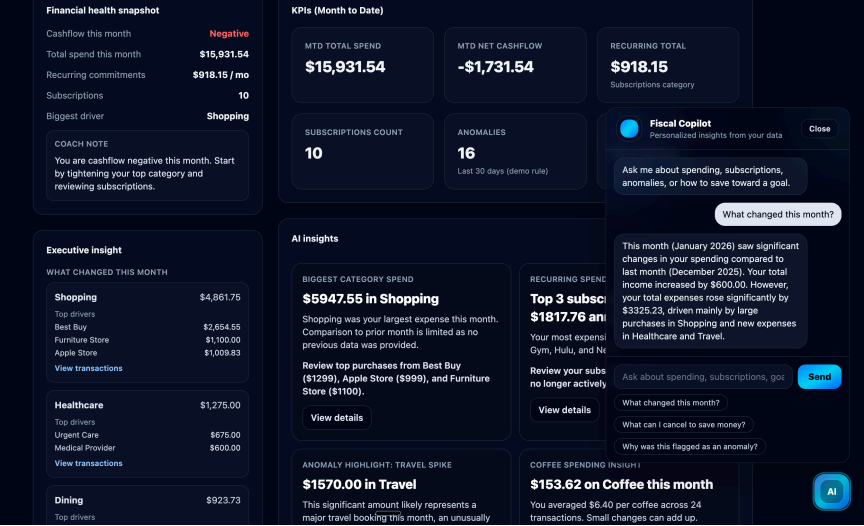
- **What the user can do**
 - View up to 5 AI generated insight cards for the selected month.
 - Open a card for details and trigger a drilldown:
 - Category drilldown loads the same category breakdown data shown on the dashboard.
 - Subscriptions drilldown loads recurring charges and highlights top cancel candidates.
- **Reliability behavior**
 - Insights are cached in memory with a TTL.
 - If the model is rate limited or fails, the system returns cached insights if available.
 - If no cache exists yet, it returns deterministic offline fallback cards generated from the loaded dataset.

- **API used**
 - `GET /api/copilot/insights?month=YYYY-MM`



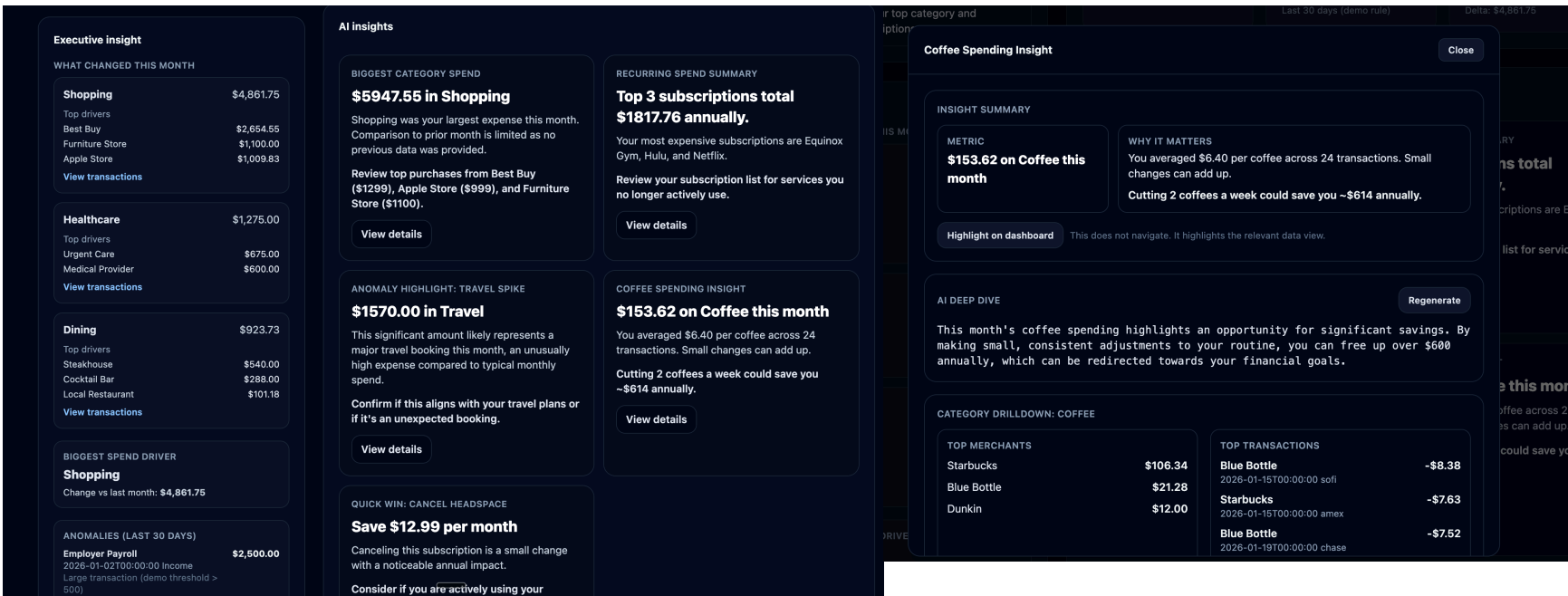
3.10 Fiscal Copilot chat

- **What the user can do**
 - Ask freeform questions in a persistent chat widget (with quick prompt chips).
 - The backend answers using the most recent transaction rows from the in memory dataset.
- **API used**
 - `POST /api/copilot/chat` with `{ message, max_rows }`



3.11 AI deep dive expansion inside insight details

- **What the user can do**
 - From an insight card's details modal, request an expanded action plan (and regenerate it).
- **How it works**
 - The UI builds a structured prompt from the selected card fields plus the active month and calls the same copilot chat endpoint to generate a longer, more actionable narrative response.
- **API used**
 - `POST /api/copilot/chat`



4. System overview

4.1 High level architecture (diagram description)

Client (React SPA, Parcel dev server on 5173)

- Routes:
 - `/` Dashboard
 - `/subscriptions` Recurring charges
- Shared shell: `AppLayout` (navigation + persistent UI)
- Key UI modules:
 - Dashboard KPIs, charts, anomalies list
 - AI Insight Cards grid with modal deep dive + drilldowns
 - Savings Goal card (stored in browser localStorage)
 - Floating Fiscal Copilot chat widget

API Layer (Flask on 5001, CORS enabled, `/api` prefix)

- Blueprints:
 - `main` (`/health` , `/seed` , `/stats` , `/transactions`)
 - `dashboard` (`/dashboard/summary` , `/dashboard/charts` , `/dashboard/category-breakdown` , `/dashboard/insights/monthly-deltas` , `/dashboard/anomalies`)
 - `subscriptions` (`/subscriptions`)
 - `copilot` (`/copilot/insights` , `/copilot/chat`)

Data and analytics (in process)

- In memory data store: `STORE`
 - `STORE.accounts` (account metadata)
 - `STORE.transactions` (merged pandas DataFrame)
- Analytics services (pandas groupby + transformations)
 - KPI summary + chart series
 - Category drilldowns
 - Month over month deltas (requires at least 2 months)
 - Rule based anomalies
- Recurring detection service
 - Merchant level cadence + confidence scoring
 - Flags for trial to paid and price increase

AI provider (server side)

- Gemini via `google.genai.Client` using `GEMINI_API_KEY` or `GOOGLE_API_KEY`
- Used in two places:
 - `/api/copilot/insights` generates 5 structured insight cards with caching and deterministic fallback
 - `/api/copilot/chat` answers user questions using recent transactions (row capped)

Text diagram (how components connect)

React UI (5173)

→ `/api/*` proxied to Flask (5001)

→ Flask routes call services

- Services read `STORE.transactions` (pandas)
 - Optional: call Gemini for insights or chat
 - JSON back to UI for rendering and drilldowns
-

4.2 Data flow (diagram description)

1) Demo ingestion

- User triggers demo load from the UI
- Frontend calls `POST /api/seed`
- Backend `seed_demo_data()` loads three repository CSVs:
 - `data/american_express.csv` → account_id `amex`
 - `data/sofi.csv` → account_id `sofi`
 - `data/chase_bank.csv` → account_id `chase`
- Each CSV is normalized into a shared schema (`normalize_transactions`)
- DataFrames are concatenated, de duplicated (by `transaction_id` when present), and stored in memory as `STORE.transactions`
- Backend returns `GET /api/stats` style summary as the seed response

2) Dashboard analytics pipeline

- Dashboard loads KPIs via `GET /api/dashboard/summary`
 - Uses `STORE.transactions` to compute month to date totals, net cashflow, recurring total for Subscriptions category, subscriptions count, anomalies count, and biggest spend driver logic
- Charts load via `GET /api/dashboard/charts?month=YYYY-MM`
 - Backend returns:
 - `available_months`
 - `spend_by_category_month`
 - `in_vs_out_month`
 - `daily_spend_trend`
- Optional drilldowns used by the UI
 - `GET /api/dashboard/category-breakdown?month=...&category=...`
 - `GET /api/dashboard/insights/monthly-deltas?month=...`

3) Anomalies flow

- Dashboard calls `GET /api/dashboard/anomalies?days=30&limit=10`
- Backend applies deterministic rules on outgoing transactions only:
 - Only `amount < 0`
 - Only within the last N days (relative to max date in data)
 - Only large charges (`abs(amount) > 500`)
 - Flags if merchant is first time seen or merchant had a gap greater than 30 days
 - Optional price spike check when there is enough history

4) Recurring charges flow

- Subscriptions page calls `GET /api/subscriptions?min_occurrences=2`
- Backend groups by merchant, infers cadence, computes confidence, annualized cost, and flags (trial to paid, price increase)
- UI provides filters (merchant search, cadence, minimum confidence) and renders a table

5) AI insights and Copilot flow

- AI Insight Cards grid calls `GET /api/copilot/insights?month=YYYY-MM`
 - Backend builds a strict JSON prompt using recent transactions (row capped)
 - Caches results in memory with TTL
 - On Gemini quota or failure, returns cached insights if available, otherwise deterministic fallback cards
 - Always returns exactly 5 cards
 - Insight details modal triggers a deep dive by calling `POST /api/copilot/chat`
 - The modal sends a structured prompt derived from the selected card
 - Backend includes recent transactions (row capped) to ground the response
 - Floating Fiscal Copilot widget also uses `POST /api/copilot/chat` for ad hoc questions
-

4.3 Key design tradeoffs

Speed vs accuracy

- Decision: in memory pandas analytics and heuristics, with “today” derived from the maximum `posted_date` in the dataset
- Why: fastest iteration for a hackathon prototype, predictable demo behavior with static CSVs

- Impact:
 - Very fast computations for KPIs and charts
 - Analytics correctness depends on normalization quality and available history
 - Month over month deltas can fail when only one month exists (explicit error path)

Rules vs ML

- Decision: deterministic rules for anomalies and recurring detection, LLM for narrative insights and Q and A
- Why: rules are explainable and stable for core signals; LLM adds natural language coaching without building a full ML pipeline
- Impact:
 - Anomalies are intentionally conservative (large outgoing charges only), reducing noise but missing smaller unusual events
 - Recurring detection produces interpretable fields (cadence, confidence, annualized cost, flags)
 - AI output is constrained to strict JSON and backed by caching and fallback to keep the UX resilient

Cost vs capability

- Decision: Gemini is used only for high value interactions (monthly insight cards and chat), with row limits and caching
- Why: keep the demo responsive and reduce repeated calls while still showcasing AI value
- Impact:
 - `/copilot/insights` uses TTL caching and returns cached or fallback results when rate limited
 - `/copilot/chat` clamps `max_rows` to a safe upper bound to limit payload size and cost
 - When Gemini quota is exhausted, the UI still works because core analytics and recurring detection are non AI and AI insights degrade gracefully

5. Analytics and insights engine

This section describes the analytics pipeline implemented in the backend services. All computations run in process over `STORE.transactions` (a pandas DataFrame seeded from the demo CSVs). There is no external database or offline batch pipeline in the current prototype.

5.1 Module boundaries

analytics_service.py

- Dashboard summary KPIs
- Dashboard charts (category totals, monthly in vs out, daily spend)
- Category drilldowns (top merchants, top transactions)
- Month over month deltas (category and merchant contributors)
- Rule based anomalies

recurring_service.py

- Recurring charge detection grouped by exact merchant string
- Cadence inference
- Confidence scoring
- Gray charge flags: trial to paid, price increase

copilot.py fallback insights

- Deterministic insight cards generated from the same transaction aggregates when Gemini is unavailable or rate limited

5.2 KPI computations

Demo notion of “today” and MTD window

- `today` is derived from the dataset: `today = df["posted_date"].max()`
- Month to date start: `mtd_start = today.replace(day=1)`
- MTD calculations use `df_mtd = df[df["posted_date"] >= mtd_start]`

KPIs returned by `get_dashboard_summary()`

- **MTD Total Spend**
 - Definition: total outgoing spend in the MTD window
 - Implementation: sum of negative amounts in `df_mtd`, multiplied by `1` to display as positive
 - `mtd_total_spend = sum(amount < 0) * -1`
- **MTD Net Cashflow**
 - Definition: total net across all transactions (income minus expenses) in the MTD window
 - Implementation: `mtd_net_cashflow = df_mtd["amount"].sum()`
- **MTD Recurring Total**
 - Definition: MTD spend specifically for transactions categorized as `"Subscriptions"`
 - Implementation: filter `category == "Subscriptions"` and `amount < 0`, sum, multiply by `1`

- **Subscriptions Count**
 - Definition: number of unique subscription merchants across the full dataset (not limited to MTD)
 - Implementation: `nunique(merchant)` where `category == "Subscriptions"` and `amount < 0`
- **Anomalies Count (last 30 days)**
 - Definition: number of anomalies produced by the anomaly rules over the last 30 days
 - Implementation: calls `get_anomalies(days=30, limit=5000)` and counts returned rows
- **Biggest Spend Driver (category delta vs previous month)**
 - Definition: category with the largest increase in expenses compared to the previous month (expenses only)
 - Implementation:
 - derive `month = posted_date.to_period("M")`
 - compute total expenses per category for current month and previous month using `abs(sum(amount))`
 - delta is `(cur - prev).fillna(cur)` and the max category is returned

5.3 Trend detection (monthly and daily aggregation)

The current prototype provides monthly and daily trends. Weekly aggregation is not implemented.

Spend by category for a selected month

- Input: `month` resolved via `_resolve_month()` (must be one of `available_months`)
- Data:
 - filter: `(month == selected_month) AND (amount < 0)`
 - groupby: `category`
 - metric: `abs(sum(amount))`
- Output shape: list of `{ category, value }` sorted descending

Money in vs money out by month (across all months)

- `money_in` : groupby month where `amount > 0` , sum
- `money_out` : groupby month where `amount < 0` , `abs(sum)`
- Output per month: `{ month, money_in, money_out, net = money_in - money_out }`

Daily spend trend for the selected month

- filter: `(month == selected_month) AND (amount < 0)`
- derive: `day = posted_date.date`
- groupby: `day` , `abs(sum(amount))`
- Output shape: list of `{ day, spend }` sorted by day

5.4 Subscription detection logic (recurring_service)

Recurring detection is computed by grouping transactions by `merchant` exact string match, then inferring cadence from the median day gap.

Input filtering

- Required fields: `posted_date` , `merchant` , `amount`
- Date parsing: `posted_date = to_datetime(errors="coerce")` , invalid rows dropped
- Rows considered:
 - If `include_zero_trials=True` : keep `amount <= 0`
 - Else: keep `amount < 0`
- Amounts are transformed into absolute values: `abs_amount = abs(amount)`

Cadence inference

For each merchant group (sorted by posted_date):

- compute gaps: `gaps = diff(posted_date).dt.days`
- compute median gap (`med`) and std dev (`std`)
- match against cadence rules using absolute tolerance:

Cadence	Target days	Tolerance days	Annual multiplier
weekly	7	3	52
biweekly	14	4	26
monthly	30	10	12
annual	365	20	1

If the merchant’s median gap does not fall within any tolerance window, the merchant is not considered recurring.

Amount tolerance and stability filter

- For weekly and biweekly merchants, if there are at least 3 paid charges, the service filters out highly variable series using coefficient of variation:
 - stable if `std(amounts) / mean(amounts) <= 0.20`
 - unstable weekly or biweekly merchants are excluded

Output metrics per recurring merchant

- `avg_amount` : mean of paid amounts (handles trial to paid by excluding the trial value from the average)
- `last_charged_date` : last posted date
- `occurrences_count` : number of charges for the merchant
- `annualized_cost` : `avg_amount * annual_multiplier`

Confidence scoring

- Regularity factor:
 - if `std_gap_days <= tolerance_days` : regularity = 1.0
 - else: regularity = 0.6
- Confidence:
 - `confidence = min(1.0, round((occurrences / 6.0) * regularity, 2))`

Results are sorted by `annualized_cost` descending.

5.5 Gray charge detection logic (current flags)

The prototype's gray charge logic is implemented as two explicit flags returned under `flags`.

Flag: trial_to_paid

Computed over the absolute amounts sequence for a merchant:

- If the first charge is near zero (`<= 1.0`) and the median of subsequent charges is at least `>= 5.0` , flag true
- Or, if the first charge is non zero and later median is at least `2.5x` the first charge, flag true

Flag: price_increase (strict)

Only evaluated for cadence in `monthly` or `annual` , and only if the series is stable:

- Stability requirement for price checks:
 - coefficient of variation `<= 0.12` on the paid series
- Last two charges must be close (indicating a real new price):
 - `abs(last - prev) / prev <= 0.05`
- Price increase condition:
 - median(second half) exceeds median(first half) by at least 15 percent and at least \$2

No other gray charge categories (like low value recurring) are computed in the current prototype.

5.6 Anomaly detection logic (analytics_service)

Anomalies are intentionally conservative and fully rule based.

Candidate filtering

- Only outgoing transactions: `amount < 0`
- Only in a recent window: last `days` relative to `max(posted_date)` in data
- Only large charges: `abs(amount) > 500`

Merchant history features

For each merchant, transactions are sorted and the following are computed:

- `prev_posted_date` per merchant using shift
- `gap_days` = days since previous outgoing charge for that merchant
- `occ_idx` = occurrence index per merchant (0 for first seen)
- `past_median_abs` = expanding median of prior abs amounts (shifted by 1)
- `past_count` = number of prior charges for that merchant

Flag rules (a candidate is an anomaly if any are true)

- First ever outgoing charge for the merchant**
 - `occ_idx == 0`
- Reappeared after a long gap**
 - `gap_days > 30`
- Price spike vs merchant history (optional rule)**
 - Requires at least 3 prior points and a non zero past median

- spike if:
 - `cur_abs >= max(500, past_median * 2)` and
 - `(cur_abs - past_median) >= 200`

Returned explanation

Each anomaly includes a human readable `reason` string:

- first time merchant
- gap since last charge
- much larger than typical, includes an approximate multiplier

Results are sorted by largest absolute amount, then newest.

How this can evolve toward ML (future)

The current system is deterministic. A next step would be to keep the same API contract while swapping the scoring layer:

- Train per merchant or per category anomaly scores using historical distributions (z score, robust z score, isolation forest)
- Add seasonality context (day of week, month) before flagging
- Learn dynamic thresholds per user and merchant rather than fixed `$500` cutoff
- Keep reason generation by attaching top contributing features to each flag

6. Backend overview

Architecture

- Flask app built using the **application factory pattern** in `backend/src/__init__.py` (`create_app`), which centralizes configuration, CORS, and blueprint registration.
- Routes are organized using **Flask Blueprints** and aggregated under a single `api_bp` in `backend/src/routes/__init__.py`, then mounted at `/api`.

Most important backend modules

- `backend/src/services/store.py`
 - Defines `InMemoryStore` and a singleton `STORE` that holds:
 - `accounts` dict
 - `transactions` pandas DataFrame
- `backend/src/utils/normalize.py`
 - Validates and normalizes incoming CSV DataFrames into a canonical schema
 - Adds `account_id` and `direction` fields
- `backend/src/services/ingestion_service.py`
 - Seeds and loads the demo CSV files into `STORE`
- `backend/src/services/analytics_service.py`
 - Computes dashboard summary, charts, month deltas, category drilldowns, anomalies
- `backend/src/services/recurring_service.py`
 - Detects recurring charges (subscriptions) and gray charge flags
- `backend/src/services/gemini_client.py` and `backend/src/routes/copilot.py`
 - Wrap Gemini calls and expose `/api/copilot/insights` and `/api/copilot/chat`, including caching and fallback behavior

Design choices that keep it modular

- Routes are thin and delegate logic to service modules.
- Shared state is centralized in `STORE` so services do not pass large DataFrames across layers.
- Normalization is a dedicated utility so ingestion and analytics can assume consistent column types.

7. Frontend overview

Architecture

- React app with client side routing defined in `frontend/src/App.jsx` using `react-router-dom`.
- `AppLayout` acts as the shared shell for all pages and hosts common UI elements.

Pages and key components

- Pages:
 - `frontend/src/pages/Dashboard.jsx`
 - `frontend/src/pages/Subscriptions.jsx`
- Shared UI:
 - `AppLayout`, `AppHeader`, `Error`
 - `CopilotWidget` (persistent floating chat)
 - `InsightCards` (AI insight cards plus modal deep dive)

- `SavingsGoalCard` (goal input stored in browser)
- Charts are isolated in `frontend/src/components/charts/` to keep the Dashboard page readable.

API integration pattern

- `frontend/src/api/client.js` provides a small fetch wrapper with:
 - JSON defaults
 - consistent error handling
 - a single base URL (`REACT_APP_API_BASE_URL`)
- `frontend/src/api/dashboardApi.js` is a purpose specific wrapper that maps each backend route to a function, keeping query string logic out of UI components.

8. Data overview

Canonical transaction schema

Normalization enforces required columns and outputs a canonical set of fields in `backend/src/utls/normalize.py` .

Required input columns:

- `transaction_id`
- `posted_date`
- `merchant`
- `amount`
- `currency`
- `category`

Canonical output columns stored in memory:

- `transaction_id`
- `posted_date` (parsed, stored as date)
- `merchant` (trimmed and whitespace normalized)
- `amount` (numeric)
- `currency` (defaults to `"USD"` if missing)
- `category` (defaults to `"Uncategorized"` if missing)
- `account_id` (set by ingestion, identifies which seeded CSV it came from)
- `direction` (`"expense"` if amount < 0 else `"income"`)

Storage model

- `STORE.transactions` is a pandas DataFrame containing the full normalized dataset for the current process lifetime.
- `STORE.accounts` is a dict keyed by `account_id` holding basic account metadata loaded during seeding.

9. Design choices

Maintainability patterns

- **Application factory pattern:** backend initializes via `create_app` , which keeps config, CORS, and route registration centralized and easy to extend.
- **Modular UI:** frontend pages are thin and call an API wrapper layer, while charts and widgets are isolated into components for readability and reuse.

Clear separation of concerns

- **Routes are thin controllers:** Flask Blueprints handle HTTP parsing and response formatting, then delegate all business logic to service modules (`analytics_service` , `recurring_service` , `ingestion_service` , `gemini_client`).
- **Service layer owns computation:** analytics and detections are implemented as pure functions over a normalized pandas DataFrame, which keeps logic testable and reusable across endpoints.
- **Single in memory source of truth:** `STORE` centralizes `accounts` and `transactions` so every route reads consistent state without duplicating ingestion or recomputation logic.

Reliability and safety first for AI

- **Fail safe behavior:** AI endpoints never break the UI. `/copilot/insights` guarantees 5 cards by using caching, last known good snapshots, and deterministic fallback cards derived from transaction aggregates.
- **Strict JSON contract:** model outputs are parsed using a defensive JSON parser that tolerates minor formatting noise and rejects invalid payloads.
- **Cost control guardrails:** `max_rows` is clamped server side to avoid oversized prompts and unpredictable latency.

10. Technical stack

Backend

- **Language:** Python
- **Framework:** Flask
- **Data processing:** pandas (all analytics, grouping, and transformations)
- **CORS:** flask-cors
- **Config:** python-dotenv
- **AI:** `google-genai` (Gemini)
 - Model is controlled by `GEMINI_MODEL` env var (default in code: `gemini-2.5-flash`)
 - API key via `GEMINI_API_KEY` or `GOOGLE_API_KEY` env var

Frontend

- **Language:** JavaScript
- **Framework:** React
- **Routing:** react-router-dom
- **Charts:** Recharts
- **Styling:** Tailwind CSS (via PostCSS)
- **Bundler / dev server:** Parcel (dev on port 5173)

11. Potential future enhancements

Secure bank connectivity

- Integrate **Plaid** (or equivalent) to connect real bank and credit card accounts.
- Support OAuth style linking via a backend generated Link token and a public token exchange flow.
- Store only the minimum required identifiers:
 - access token stored encrypted server side
 - account ids and item ids
 - no raw credentials stored in the app
- Implement transaction sync:
 - initial backfill after link
 - incremental sync using cursor based pagination
 - webhooks to refresh when new transactions arrive
- Add account management:
 - link multiple institutions
 - relink on credential rotations
 - remove account and delete all associated data

Production grade storage and data lifecycle

- Replace in memory store with persistent storage:
 - tables for users, institutions, accounts, transactions, recurring merchants, anomaly events, cached insights
- Data retention controls:
 - user driven delete account and delete all data
 - configurable retention window for raw transactions
 - store aggregates and derived insights separately from raw rows
- Encryption and secrets:
 - encryption at rest for sensitive fields
 - rotate keys and tokens
 - strict separation between app secrets and user tokens

Analytics engine upgrades

- Expand trend analytics:
 - weekly aggregation and seasonality views
 - rolling spend velocity and budget pacing
 - income variability metrics for gig and freelance users
- Improve “what changed” explanations:
 - distinguish price increases vs frequency increases vs new merchants
 - attribution to top merchants with delta decomposition
- Category quality:
 - automatic category normalization

- user overrides that persist and reflow through all analytics

Subscriptions and gray charges upgrades

- Merchant normalization and fuzzy grouping so the same merchant variants merge correctly.
- Detect more recurrence patterns:
 - quarterly, semiannual, irregular series
- Additional gray charge flags:
 - subscription clutter (many low cost recurring)
 - duplicate services in same category
 - silent price creep detection with confidence scoring
- Optional cancellation assistance:
 - generate cancel steps or links only when verified sources exist
 - export to CSV

Anomaly detection evolution

- Move from fixed thresholds to adaptive scoring:
 - per merchant robust z score
 - isolation forest style outlier detection
 - category baseline models
- Add anomaly types beyond large charges:
 - duplicate charges in a short window
 - unusual frequency spikes
 - refund reversals and chargebacks

Goal tracking and forecasting

- Add goal creation and tracking:
 - target amount and target date
 - monthly required savings
- Baseline forecasting:
 - rolling averages for income and spend
 - scenario planning using subscriptions and category cuts
- More advanced forecasting:
 - seasonality models
 - separate fixed vs variable expenses

AI copilot enhancements

- Ground AI outputs primarily on computed aggregates rather than raw rows to reduce hallucinations and cost.
- Add tool style routing:
 - force AI to select from known drilldowns (subscriptions, anomalies, category breakdown)
- Add evaluation harness:
 - golden prompt set
 - schema validation tests
 - regression tests for fallback behaviors

Deployment and operations

- Docker Compose setup for backend, frontend, and database.
- CI pipeline:
 - lint, format, unit tests, smoke tests
- Observability:
 - request latency
 - ingestion and sync success rates
 - AI error rate and cache hit rate
 - alerting on webhook failures