

# Budget Viewer Module Implementation Guide and Specification

## 1. Budget Definition

The Budget Viewer module revolves around a **Budget** entity representing a planned spending limit for a specific category and time period. Each user can have multiple budgets (e.g. separate category budgets for groceries, rent, etc.), covering both personal and business categories. Below we define the schema for budgets in both SQL and JSON, including all required fields:

### Budget Schema (SQL)

```
CREATE TABLE budgets (  
  id SERIAL PRIMARY KEY,  
  userId INTEGER NOT NULL REFERENCES users(id),  
  categoryId INTEGER NOT NULL REFERENCES categories(id),  
  period VARCHAR(20) NOT NULL, -- e.g. 'monthly', 'weekly', 'custom'  
  amount DECIMAL(12,2) NOT NULL, -- budget limit in currency  
  startDate DATE NOT NULL, -- start of first budget period  
  endDate DATE NULL, -- optional end date if budget is  
  temporary  
  rollover BOOLEAN NOT NULL DEFAULT FALSE, -- whether to carry over  
  balance  
  thresholds JSON NOT NULL DEFAULT '[80,100,120]' -- alert thresholds (%)  
  as array  
);
```

### Field explanations:

- `userId` ties the budget to a specific user (supporting multiple budgets per user).
- `categoryId` links to a spending category the budget applies to.
- `period` defines the recurrence interval of the budget (e.g. monthly, weekly, or custom range). For recurring budgets, the system will generate a new budget period snapshot at each interval.
- `amount` is the budgeted spending limit for each period.
- `startDate` and `endDate` define the active date range of the budget. For monthly/weekly budgets, `startDate` is the reference point (e.g. the first day of the first month/week) and `endDate` can be null for an ongoing budget or set for a custom one-time budget.
- `rollover` indicates if unspent money should carry over to the next period (and similarly if overspend should reduce the next period's allowance). If `rollover=true`, any remaining budget balance (positive or negative) at period end is added into the next period for that category <sup>1</sup> <sup>2</sup>. This is useful for categories with irregular or “lumpy” expenses (e.g. annual fees, gifts) so that

leftover funds accumulate for future large expenses <sup>1</sup>. Overspending can also carry over as a deficit to the next period <sup>2</sup>.

- `thresholds` defines usage levels (as percentages of the budget amount) at which alerts are triggered. By default we use 80%, 100%, and 120% (meaning 20% over budget) as key thresholds. These can be stored as a JSON array of percentages (or could be a JSON object with custom values per budget). For example, `[80, 100, 120]` means warn at 80% usage, at 100% (budget fully used), and at 120% (significant overspend). These thresholds can be customized per budget in the future if needed (some budgets might have no alerts or different values).

## Budget Representation (JSON)

When communicating budget data via an API or configuring it in code, a JSON representation is useful. Below is an example JSON object for a budget entity:

```
{
  "id": 101,
  "userId": 42,
  "categoryId": 7,
  "period": "monthly",
  "amount": 500.00,
  "startDate": "2025-01-01",
  "endDate": null,
  "rollover": true,
  "thresholds": [80, 100, 120]
}
```

This example defines a monthly budget of \$500 for category 7 (e.g. Groceries) starting Jan 1, 2025, with rollover enabled and default threshold alerts at 80%, 100%, 120% usage.

## 2. Budget Calculation

Budget calculation involves determining how much has been spent in each category for the current period, handling carry-over balances if rollovers are enabled, and adjusting for any irregular transactions. The system must aggregate expenses in real-time (or near-real-time) to show up-to-date budget utilization.

### Expense Aggregation by Category and Period

For each budget, we need to sum all expenses (transactions) that fall into the budget's category and period timeframe. This can be done via a query grouping transactions by category and filtering by date range. The *current period* for a budget can be determined based on the budget's `period` and `startDate`:

- **Monthly budgets:** Calculate the current month period boundaries. If `startDate` is, say, Jan 1, the current period would be the calendar month containing today's date (e.g. June 1 – June 30, 2025 for June 2025). If a budget started mid-month, we treat that as the first period boundary.
- **Weekly budgets:** Similarly, determine the week range (e.g. if `startDate` is a Monday, each period is Monday–Sunday).

- **Custom date range:** If `period` is custom (one-off budget), use the given `startDate` - `endDate` as the budget period.

**Summing expenses:** For a given budget and period, sum all transaction amounts where `transaction.userId = budget.userId`, `transaction.categoryId = budget.categoryId`, and `transaction.date` falls between the period's start and end (inclusive). This sum is the **period spending (spent)**.

Additionally, for accuracy: - Include **adjustments for refunds:** if a transaction is a refund or credit in that category (negative expense), it should subtract from the spent amount. - If the budgeting app supports **transfers or credits** that affect budgets, handle accordingly (though typically budgets track expenses only).

## Rollover Logic (Under- and Overspend Carryover)

If `rollover` is enabled for a budget, the remaining balance from the previous period must be applied to the current period. There are two cases: - **Underspend (surplus):** If the user spent less than the budget in the last period, the unspent amount is added to the new period's budget. For example, if last month's grocery budget was \$500 and only \$400 was spent, \$100 carries over. This effectively makes the new period's available budget  $\$500 + \$100 = \$600$  <sup>3</sup> (assuming rollover to the same category). - **Overspend (deficit):** If the user exceeded the budget in the last period (e.g. spent \$550 on a \$500 budget, \$50 over), that amount is subtracted from the new period's budget <sup>2</sup>. So the next period's effective budget would be  $\$500 - \$50 = \$450$ , since the overspent \$50 needs to be "covered" <sup>2</sup>. This helps enforce accountability by not resetting budgets to full amount after an overspend.

To implement this, we maintain a running **carryover balance**. One approach is to compute the **remaining balance** at the end of each period: `remaining = budget.amount - spent` for that period (this can be positive, zero, or negative). When a new period starts, adjust the budget: - If rollover enabled: `effectiveBudget = budget.amount + previous_remaining`. (If the previous\_remaining was negative, this effectively reduces the budget.) - If rollover disabled: `effectiveBudget = budget.amount` (each period stands alone, no carryover).

**Example:** Suppose a quarterly budget of \$3000 for a category, rollover enabled: - Q1: Spent \$2500, remaining +\$500 → Q2 budget becomes  $\$3000 + \$500 = \$3500$ . - Q2: Spent \$3600 (which is \$100 over the \$3500) → remaining -\$100 → Q3 budget =  $\$3000 - \$100 = \$2900$ . - Q3: ... and so on, carrying forward cumulatively. (Rollover balances accumulate across periods if not used <sup>2</sup>.)

The carryover amounts should be stored or derivable. We will store period snapshots (discussed later) that include spent and remaining, which can be referenced for the next period calculation.

## Adjusting for Refunds and Backdated Transactions

Real-world finances can be messy: refunds may occur, and transactions might be recorded late (or re-categorized). The budget calculation must handle these gracefully:

- **Refunds:** If an expense in a category is refunded within the same period, it will appear as a negative transaction. This reduces the spent amount. Our aggregation should naturally handle this by summing negatives with positives. If a refund comes in after the period closed, we should

retroactively adjust the historical period's records (and possibly the carryover). For example, if you returned a \$50 item that was counted in last month's budget, that \$50 should be added back to last month's remaining balance (and thus increase the carryover into this month if rollover applies).

- **Backdated or late transactions:** A transaction dated in a previous period might only get imported/entered now. In a rollover scenario, this affects the previous period's actual spend and remaining balance, which in turn affects the carryover into all subsequent periods. To handle this, whenever new transactions are ingested, we should **recompute the affected budgets**:
- If a transaction's date falls in a past budget period (e.g. last month), recalc that period's spent and remaining.
- Then propagate any change in remaining to the next period's effective budget (if rollover), and update subsequent period snapshots as needed.
- This can be done in the nightly job or immediately upon transaction import (more on Ops in section 6).

By continuously aggregating expenses and updating for any changes, the Budget Viewer can present an accurate picture of spending vs. budget at all times.

### 3. Alerts & UX

A core feature of the Budget Viewer is to alert users as they approach or exceed their budget limits. This involves defining threshold levels, presenting visual indicators in the UI, and delivering notifications through various channels. The aim is to **proactively warn users** so they can adjust spending behavior (as one user noted, getting an 80% budget warning helped them realize "I gotta stop going to Wendy's" before blowing the budget <sup>4</sup> ).

#### Threshold Alerts (80%, 100%, 120%)

We define three default alert thresholds for budget usage: - **80% of budget (Warning)** – signifies the user has used a significant portion of the budget. At this point, we flag the budget in **yellow** and send a gentle warning that, for example, *"You've spent 80% of your Dining Out budget."* Many budgeting systems use a similar threshold (e.g. cloud cost monitors often set 80% as a warning level) <sup>5</sup> . This early warning gives users a chance to cut back before it's too late <sup>4</sup> . - **100% of budget (At Limit)** – the user has fully exhausted the budget. This is a critical alert (mark in **red**), e.g. *"Budget maxed out: you've reached 100% of your \$500 budget."* The UI will prominently highlight this budget in red, indicating no room is left. A notification is sent at the moment the budget crosses from 99% to 100% or above. - **120% of budget (Overrun)** – the user has exceeded the budget by 20% or more. This is an **over-budget alert** indicating a significant overspend, e.g. *"You are at 120% of your budget (20% over)!"* We choose 120% as a threshold to alert on overshoot beyond a small buffer. At 120% (or any configurable overspend percentage), we can send a stronger alert or even escalate (perhaps mark as **"Overspent"** status). In practice, some systems use 110% as an overspend alert <sup>5</sup> ; our default of 120% can be adjusted based on user preference or forecast (see below).

These thresholds are configurable per budget via the `thresholds` field. Some budgets might disable certain alerts or use custom percentages (e.g. a user could set a very strict alert at 50% for a discretionary category, or no alerts for a fixed expense category). The system should allow **custom threshold configuration per budget item** <sup>6</sup> in the future, but 80/100/120 serve as sensible defaults.

**Alert triggering logic:** The application will monitor the budget utilization. When a threshold is crossed (e.g. the moment spent amount  $\geq 0.8 * \text{amount for } 80\%$ ), it triggers an alert event. Importantly, to avoid spamming, each threshold should trigger at most **one alert per period**. For example, if a user oscillates around 80% (due to refunds, etc.), we don't want to send multiple 80% alerts – just one when it's first crossed. Similarly, the 100% alert is only sent once when the budget is hit/exceeded. The 120% alert only triggers if spending continues to rise to that level (and again, only once). We will maintain state of which alerts have been sent for the current period (this could be tracked in the `budget_snapshots.status` or a separate alerts log).

## Visual UX Components (Indicators and Progress Bars)

In the app's interface (built with Next.js/React and Tailwind CSS), budgets will be displayed with intuitive visual indicators:

- **Progress Bars:** Each budget can be represented by a horizontal progress bar showing the percentage of spending. For example, a bar that fills from 0% to 100% in green/yellow/red as the budget is utilized. We can implement this with a simple `<div>` styled in Tailwind: a grey background bar with a colored inner bar whose width is `spent/amount * 100%`. The color changes based on thresholds (e.g. class changes from `bg-green-500` to `bg-yellow-500` at 80%, and `bg-red-500` at 100%). We will also overlay a label like "75%" or "\$375 of \$500" for clarity.
- **Circular "donut" gauges:** For a more compact or visually engaging component, a circular progress indicator (dial) can show budget usage. The circle would be colored similarly (green/yellow/red) and possibly display the percent in the center. This could be implemented using an `<svg>` circle or a Canvas, or by utilizing existing Tailwind UI components (e.g. DaisyUI's `radial-progress`). The circular format is useful for dashboards and is immediately recognizable as a gauge of how much budget is left.
- **Color Coding (Green/Yellow/Red):** As mentioned, we use color cues to convey status:
  - **Green:** safely under budget (e.g. less than 80% used). This is the normal state.
  - **Yellow:** caution, approaching budget (after 80% used). Draws the user's attention.
  - **Red:** budget reached or exceeded (100%+). This indicates action is needed. In research on spending tools, a tachometer-style indicator with green/yellow/red zones is effective at showing whether the user's current spending pace is reasonable or excessive <sup>7</sup>. We emulate that concept with our color zones.

In addition to bars and gauges, we might display icons or text:

- For example, a small ⚠ icon or "Almost there" text when in yellow zone, and a 🚫 or "Over budget" label when in red zone.
- A numeric "remaining" value (e.g. "\$50 left" or "\$50 over") can be shown alongside the bar to explicitly tell the user how much remains or how much they've exceeded.

These components will be styled with Tailwind for consistency. For instance, a budget card might use classes like `bg-red-100` or `bg-yellow-100` as a light background tint for categories that are over or near budget, to make them stand out in the list.

## Alert Delivery Channels

When thresholds are crossed, the system should deliver notifications through multiple channels to ensure the user gets the alert in their preferred medium. The supported channels will include:

- **Email (Gmail):** Send an email notification to the user. For example, using a service or SMTP to send a templated email: "Subject: Budget Alert – 80% of Groceries Budget Used".
- **Telegram:** If the user links their Telegram account or bot, we can push a message via Telegram's Bot API. The message might read " *Budget Alert:* You've spent 80% of your Groceries budget".

- **WhatsApp:** Similar to Telegram, using WhatsApp Business API or an integration (like Twilio), send a WhatsApp message alert.
- **Browser Notification:** Using the Browser Notifications API (via a Service Worker in our Next.js PWA or web app), pop up a notification on the user's desktop or device. E.g. "Budget Viewer: 100% of Travel budget reached!" which the user sees even if the web app isn't focused.
- **OS Notifications (Desktop/Mobile):** If the user has a desktop app or if they allowed system notifications via the browser, ensure the notifications appear in their OS notification center.
- **Mobile Push Notifications:** In anticipation of a future mobile app, design the alert system such that push notifications (via APNs for iOS, FCM for Android) can be sent when thresholds are hit.

To manage this, we can create a **Notification Service** that the Budget Viewer calls when an alert condition occurs. This service will take the alert event (which user, which budget, what threshold crossed, message to send) and broadcast it to the user's enabled channels. Users might have preferences (e.g. enable/disable certain channels), which we would respect.

**Example alert flow:** User crosses 100% on a budget: 1. In the nightly snapshot job (or real-time check), we detect `spent >= amount` for the first time. 2. We mark the budget's status as "exceeded" and log that 100% alert was triggered. 3. The Notification Service formats the alert message. 4. It sends out: an email, a Telegram bot message, and a push notification, as configured. 5. The next time the user opens the app, the UI might also show a red banner or highlight on that budget.

By providing multi-channel alerts, we increase the chance the user is aware of their budgeting status and can take action immediately. This proactive approach is proven to help users stay on track rather than finding out at month-end they blew the budget <sup>8</sup>.

## 4. Forecast Integration

In addition to showing current budget status, the Budget Viewer will include a **spending forecast** for each budget period. Forecasting helps answer "Where will I end up by period's end if I continue this spending behavior?" and allows early intervention if overspend is likely. We integrate a forecasting engine that supports multiple strategies and provides a projection of end-of-period spend, along with an indication of overspend risk.

### Projected Spend "At Current Pace"

The simplest forecast is a linear extrapolation of spending to date. Essentially, we assume the user will continue to spend at the same average rate for the remainder of the period: - Calculate the daily spending rate so far = `spent_so_far / days_elapsed`. - Project total period spend = `spent_so_far + (daily_rate * days_remaining)`.

For example, if today is Day 10 of 30 and \$300 has been spent, `daily_rate` = \$30/day, `days_remaining` = 20, so projected total = \$300 + \$30\*20 = \$900. If the budget was \$1000, the forecast would indicate the user is on track to be under budget (~90% used). If the budget was \$500, the forecast would show they are pacing to \$900 which is 180% of budget, a clear warning.

This "current pace" linear forecast is easy to compute and provides a baseline projection. It can be refined by weighting recent days more, excluding one-off large purchases, etc., but as a first approximation it's

valuable. We will display the projected end-of-period spend in the UI (e.g. "Projected: \$900 by 30 Sep") alongside the current spent amount.

## Advanced Forecast Strategies (Toggleable per User/Budget)

Different users might prefer different forecasting methods. Our system will support three strategies that can be toggled (globally or per budget):

1. **Simple Linear (Trend) Extrapolation:** (Default) The method described above. It may also incorporate the past periods' average to smooth out anomalies. This is essentially a *rule-based or statistical* approach using recent data. It's easy to explain and understand (good for user transparency).
2. **Machine Learning Model:** A more sophisticated approach that uses historical data and possibly other factors to predict future spending. For example, a model could learn seasonal patterns or personal habits (perhaps spending more on weekends, or spikes at month-end). A machine learning algorithm (like a regression model or even a time-series model) can analyze past spending patterns and forecast future expenses more adaptively <sup>9</sup> <sup>10</sup>. Research shows ML can detect hidden spending trends and improve forecast accuracy compared to static rules <sup>10</sup>. For instance, a Random Forest or ARIMA model could be trained per category using the user's prior months of data <sup>11</sup>. Initially, we might implement a simpler ML like linear regression on past periods, and later upgrade to advanced techniques (possibly even a small neural network or an external forecasting service).
3. **Rule-Based Forecast Configuration:** This allows the user to define or choose specific rules for forecasting. For example, a user might input known upcoming expenses (e.g. *"I plan a \$200 purchase in this category next week"* which the forecast should include), or set rules like *"assume my spending increases 10% during holidays"*. Another rule-based option is using **last period's same time** as a guide (e.g. *"by this day last month you had spent \$X, compare that to \$Y this month"*). Essentially, this mode could be a configurable formula – for advanced users who want fine control. In implementation, we could let users pick from preset rule-based options (like "use last month's total as forecast" or "use year-over-year same month growth") or define custom parameters.

The system's architecture should be flexible to add more strategies. Each budget or user profile could have a setting like `forecastStrategy` that can be one of `["linear", "ml", "rule"]`. The UI would allow toggling between these so the user can see different projections (like a tab or dropdown to switch forecast mode).

Under the hood, we might implement the ML and rule-based forecasts in a separate service or module due to complexity:

- The ML forecast might call a prediction function that either uses a pre-trained model or runs an on-the-fly regression. (Since our stack is TypeScript, we could use a library or call a Python microservice for ML, but details are beyond scope here.)
- The rule-based forecast could parse user-defined rules or simply apply configured logic (e.g. a user might choose "assume same spend as last period" which is trivial to compute from the snapshots).

## Forecast Output and Overspend Risk

Regardless of strategy, the forecast engine will output a **projected total spend for the period** (e.g. \$X by end date). Using this, we can determine the **projected vs. budget difference**: - If `projected <= budget.amount`, the user is on track (perhaps show a small green check or "On track" label). - If

`projected > budget.amount`, then overspend is likely. We calculate how much over (projected - amount) and possibly the percentage over (e.g. projected 120% of budget). This informs a risk level: - Just over budget (e.g. 105%) might be a **moderate risk** (colored orange perhaps). - Far over (e.g. 150%+) is **high risk** (colored red and maybe with an exclamation icon).

We can display a **“Projected Overspend” warning** in the UI. For example: *“Projected to exceed budget by \$400 at current pace.”* This gives the user advance notice, even before they hit the actual budget threshold alerts. It’s essentially a forecast-based alert. In fact, we could tie this into the alert system: if forecast predicts 120% usage before period end, we might trigger an alert at, say, 110% forecast to warn early <sup>12</sup>.

Visually, one idea is a **dashed line** or marker on the progress bar indicating where the projection lies. For instance, if currently 50% used but projected to 110%, we can put a red marker past the 100% end of the bar to show the overshoot. Another idea is a small tachometer-like dial (as referenced earlier) or simply text like “Projected: 110% of budget” in a highlighted style.

To summarize forecasts: - All budgets will show a projected end-of-period spend. - The user can toggle different forecast methodologies. - The system uses the chosen method to compute the projection nightly (or on-the-fly). - If projections indicate an overspend, the UI and alerts will reflect that (possibly an alert saying “At current pace, you will exceed your budget by month end”). - This forward-looking feature leverages data to help users stay within budget, embodying the idea that analyzing past patterns can *“predict future expenses based on past patterns”* <sup>9</sup> and thus allow proactive adjustments.

## 5. Data Contracts

This section formalizes the data models (SQL schema and JSON structure) for the core entities: `budgets` and `budget_snapshots`. These define the **data contract** between front-end, back-end, and any services (ensuring everyone understands the shape of the data).

### `budgets` Table Schema (SQL + JSON)

The `budgets` table was described in section 1. For completeness, here is the schema definition and an example in JSON form:

```
-- SQL Schema for budgets
CREATE TABLE budgets (
  id          SERIAL PRIMARY KEY,
  userId      INT NOT NULL REFERENCES users(id),
  categoryId  INT NOT NULL REFERENCES categories(id),
  period      VARCHAR(20) NOT NULL,    -- 'monthly', 'weekly', 'custom'
  amount      DECIMAL(12,2) NOT NULL,
  startDate   DATE NOT NULL,
  endDate     DATE NULL,
  rollover    BOOLEAN NOT NULL DEFAULT FALSE,
  thresholds  JSON NOT NULL           -- e.g. [80, 100, 120]
);
```



## JSON representation:

```
{
  "id": 101,
  "userId": 42,
  "categoryId": 7,
  "period": "monthly",
  "amount": 500.00,
  "startDate": "2025-01-01",
  "endDate": null,
  "rollover": true,
  "thresholds": [80, 100, 120]
}
```

(This JSON example is identical to the one in section 1.)

- The `thresholds` could alternatively be an object like `{"80": true, "100": true, "120": true}` or similar if we stored whether each threshold is active. But using an array of percentages is straightforward.

## budget\_snapshots Table Schema (SQL + JSON)

We introduce a `budget_snapshots` table to record computed budget status for each period. Each snapshot represents a specific budget's state at a given point (usually at the end of a day or period). This allows historical tracking and quick retrieval of current status without re-aggregating transactions every time (we update it nightly or on changes).

```
CREATE TABLE budget_snapshots (
  id SERIAL PRIMARY KEY,
  budgetId INT NOT NULL REFERENCES budgets(id),
  periodStart DATE NOT NULL,
  periodEnd DATE NOT NULL,
  spent DECIMAL(12,2) NOT NULL,      -- total spent in this period up to
now                                     --
  projected DECIMAL(12,2) NOT NULL,  -- projected total spend for period
  remaining DECIMAL(12,2) NOT NULL,  -- remaining budget (if negative,
overspent amount)
  status VARCHAR(20) NOT NULL,       -- e.g. 'green', 'warning', 'critical'
  computedAt TIMESTAMP NOT NULL      -- when this snapshot was calculated
);
CREATE INDEX idx_snapshot_budget_period ON budget_snapshots(budgetId,
periodStart);
```

- `budgetId` links to the budgets table.

- `periodStart` (and `periodEnd`) identify the budget period this snapshot is for. For example, `periodStart = 2025-06-01` and `periodEnd = 2025-06-30` for a June 2025 monthly budget. (Storing both start and end can be useful for querying; alternatively store just start if period length is implicit.)
- `spent` is the sum of expenses in that period (as of `computedAt`).
- `projected` is the forecasted total spending for the period.
- `remaining` is `budget.amount + carryover - spent` at the time of computation. It represents how much is left to spend (or negative if overspent). At period end, this value (carryover) would be rolled to next if applicable.
- `status` could be an enum or text indicating the alert state:
- For example: 'green' (under 80%), 'warning' ( $\geq 80\%$  but  $< 100\%$ ), 'critical' ( $\geq 100\%$  but  $< 120\%$ ), 'overspent' ( $\geq 120\%$ ). We could also encode these as simple states like 'OK', 'warning', 'over', etc. The status is derived from thresholds.
- `computedAt` timestamp records when we generated this snapshot (nightly job or on update). There might be multiple snapshots throughout a period (e.g. daily updates), or we might update the same row. Another design is to have one row per period per budget, updating it each day – in that case `computedAt` is just the last update time. If we keep history (say snapshots at each month-end), then we'd insert new rows per period.

#### JSON representation of a budget snapshot example:

```
{
  "budgetId": 101,
  "periodStart": "2025-06-01",
  "periodEnd": "2025-06-30",
  "spent": 450.00,
  "projected": 500.00,
  "remaining": 50.00,
  "status": "warning",
  "computedAt": "2025-06-20T00:00:00Z"
}
```

This indicates that for budget #101 in June 2025, as of June 20, the user spent \$450 out of \$500, with \$50 remaining. The projection is \$500 (meaning they are on track to exactly hit the budget by June 30). Status is "warning" since they crossed 90% (in this case 90% used) which is above the 80% threshold but haven't hit 100% yet. If they spend more and hit 100%, status would update to "critical" or "over".

These data contracts ensure that the front-end can receive structured data to render the dashboards, and the back-end knows how to store and update budget info. Any changes to the budget definition (like thresholds or amount) would reflect in these structures.

## 6. Ops and Snapshot Computation

Maintaining accurate budget data requires periodic computations and responsive updates to changes. Here we outline the operational tasks and jobs needed, including nightly snapshot recalculation, on-demand updates, and change audit trails.

## Nightly Budget Snapshot Job

We will implement a **nightly cron job** (or scheduled serverless function) that recalculates each budget's snapshot, typically in the middle of the night or early morning when load is low. The purpose of this job is to update projections and trigger alerts in a timely manner: - It iterates through all active budgets (or all budgets for which the current date falls in their active period). - For each, it computes the current period's spent amount (summing transactions, as in section 2) and updates/creates the `budget_snapshots` entry for that period. - It also calculates the forecast (projected spend) using the user's chosen strategy. - It then determines the status and checks against thresholds to see if a new alert needs to be sent.

This ensures that even if the app isn't open, the data (and alerts) stay up to date daily. (If immediate real-time updates are needed, we could also run this job more frequently or update on each transaction event, but nightly is a good baseline.)

Pseudo-code for the nightly job logic:

```
for each budget in budgets:
    periodStart, periodEnd = getCurrentPeriod(budget.period, budget.startDate)
    # Sum transactions for this budget's user + category in [periodStart, now]
    spent = sumTransactions(userId=budget.userId, categoryId=budget.categoryId,
    date between periodStart and periodEnd)

    # Apply rollover adjustment if needed
    effectiveBudget = budget.amount
    if budget.rollover:
        prevSnap = getSnapshot(budget.id, previousPeriodStart)
        if prevSnap exists:
            # If last period had remaining or deficit, adjust this period's
            budget
            effectiveBudget += prevSnap.remaining # (remaining could be
            negative or positive)
            # Note: effectiveBudget is used only for context; 'spent' already
            includes all actual spend.

            remaining = effectiveBudget - spent

    # Compute projection using chosen forecast strategy
    projected = forecastEngine.calculate(budget, transactionsSince(periodStart))

    # Determine status based on thresholds
    usagePercent = (spent / effectiveBudget) * 100
    status = "green"
    if usagePercent >= budget.thresholds[0]: status = "warning"      # crossed
    80%
    if usagePercent >= budget.thresholds[1]: status = "critical"    # crossed
    100%
```

```

    if usagePercent >= budget.thresholds[2]: status = "overspent"    # crossed
    120%

    # Update or insert snapshot
    snapshot = { budgetId: budget.id, periodStart, periodEnd, spent, projected,
    remaining, status, computedAt: now }
    upsertBudgetSnapshot(snapshot)

    # Alerting: trigger notifications for new threshold crossings
    for threshold in budget.thresholds:
        if (spent / effectiveBudget * 100) >= threshold:
            if not alertSentAlready(budget.id, periodStart, threshold):
                sendAlert(budget.userId, budget, threshold)
                logAlertSent(budget.id, periodStart, threshold)

```

In this pseudocode, `alertSentAlready` and `logAlertSent` refer to tracking so we don't duplicate alerts. The `sendAlert` function would interface with the Notification Service to actually dispatch via email/Telegram/etc.

We also handle rollover by looking at the previous period's snapshot to adjust the new period's effective budget (so overspend/underspend carries over).

## Real-time Updates on Data Changes

In addition to the nightly batch job, we need to **recompute snapshots on certain events**:

- **New transaction or transaction edit**: When a new expense is logged or an existing one is edited (amount or category changed), we should immediately update the relevant budget's spent amount. This could be done by triggering a small recomputation for that budget (and possibly subsequent ones if rollover cascades). For instance, if a transaction dated today in category X is added, update the current period snapshot for budget X accordingly. If a transaction dated last month is added, update last month's snapshot and then adjust carryover and current snapshot.
- **Budget definition change**: If the user updates a budget (e.g. changes the amount or toggles rollover), we maintain an **audit trail** of these changes. Likely, we have to reflect this going forward (and possibly retroactively in how we interpret past data).
- We can create a `budget_history` table or keep old snapshots intact. E.g., if the user increases this month's budget from \$500 to \$600 mid-month, the snapshot calculation should use the new amount for the remaining days. One approach is to apply changes only from the next period, to avoid complicating mid-period changes. Alternatively, we recalc with the new amount proportionally.
- In any case, log the change (who made it, when, old vs new values). This is important for audit and debugging ("Why did my budget status suddenly change? – because you edited the budget.").
- **Category reclassification**: If a user reassigns a transaction's category, two budgets are affected (the old category budget and the new category budget). We should recalc both. This scenario is similar to a transaction edit.

For implementing these, we can use event-driven triggers in our application:

- On a transaction insert/update/delete, call a function to update budgets.
- On a budget change, possibly recalc that budget's current snapshot from scratch (and decide how to handle historical data – probably leave historical snapshots under old rules for honesty, but mark that a change occurred).

We will also consider **late-arriving transactions** as mentioned: if our data aggregator brings in transactions a few days late, those are effectively backdated. The nightly job should be capable of looking at not just the current period but if any past period still open for changes (maybe the last 1-2 periods) and recalc them too. This is a bit more advanced, but a simple strategy is: - Always recompute the previous period as well in the nightly job, in case something changed (for a small performance cost).

## Audit Trail of Budget Changes

Maintaining an audit trail for budget definition changes can be done via: - A separate table, e.g. `budget_changes(budgetId, changedAt, oldAmount, newAmount, oldThresholds, newThresholds, changedBy, ...)`. - Or using a versioning approach: rather than updating a budget in place, we could insert a new row and mark the old one inactive (but that complicates references).

For simplicity, logging changes in a `budget_changes` table is effective. Every time a user edits a budget's properties (amount, period, rollover setting, thresholds), we insert a record with a timestamp and the delta. This doesn't directly affect calculation, but it provides a history. Our nightly job or snapshot calculation can also consult this if needed (though typically it will just use the latest values).

In summary, our Ops plan ensures the budget data is recalculated reliably each day and kept consistent with any user actions: - A scheduled job handles regular updates and alert triggers (for example, Azure's cost budgets do a daily check and send alerts within an hour of threshold breach <sup>13</sup> - our system will do similarly on a daily cadence). - Event-driven updates handle immediate changes so the UI can reflect near real-time information after a user input. - Data integrity is preserved via audit logs and recalculations, so rollovers remain accurate even as data changes.

## 7. Dashboards

Finally, we design the UI components that present all this information to the user. We have two main dashboard views: **per-category budget view** and an **all-budgets summary view**. Both will be implemented in React with Tailwind CSS, aiming for clarity and interactivity. Below we outline the component structure and technical specs for each.

### Per-Category Budget Dashboard

This view (likely a page or modal in the app) gives a detailed picture of a single budget category's status. Key elements to include:

- **Header with Category & Period:** e.g. "Groceries – June 2025 Budget". Display the budget amount and period duration (start and end dates).
- **Progress Indicator:** A prominent component showing how much of the budget is used. This could be a circular progress donut in the center or a large horizontal bar. It should be labeled with the percentage used and/or the dollar amounts (spent vs total). For example, a circular progress at 90% filled, with "90%" in the middle and maybe a colored ring (yellow since  $90\% \geq 80\%$ ). Tailwind can style this; if using a circle, we might include an SVG circle whose stroke dashoffset corresponds to remaining percentage.
- **Numeric Stats:** Below or around the progress indicator, show stats:

- **Spent:** e.g. "Spent: \$450".
- **Remaining:** e.g. "Remaining: \$50" (or "Over budget by \$X" if negative).
- **Budget:** the total budget amount (for clarity).
- These can be in a small grid or list with labels.
- **Threshold Alerts Display:** If the budget is in warning or over status, show an alert message or badge. For example, if at 120%, show a red badge "120% - Over Budget!" or an icon. This duplicates the color coding but ensures text feedback too.
- **Forecast Details:** A section that shows the projected spend. For instance: "Projected end-of-month spend: \$500 (at current pace)". If multiple forecast strategies are available, provide a toggle (maybe a segmented control or dropdown: [ Current Pace | ML Forecast | Custom Rule ]). When the user switches, update the projected number accordingly. For an ML forecast, we might label it "AI Prediction: \$540" etc. If a forecast predicts overspend, highlight that number in red and perhaps add a note "(Over budget by \$40)". We can also visualize the projection on the progress bar as mentioned (like a marker beyond 100%).
- **History or Trend (optional):** It can be useful to show how spending has progressed over the period. A small line chart or bar chart could display daily cumulative spend vs. the ideal pace. However, since this is an implementation spec, we might leave detailed chart integration for later. Still, we anticipate adding a trend chart component (perhaps using a library like Chart.js or Recharts in React) to plot spend over time and possibly the forecast curve.
- **Controls:** If applicable, controls like "Edit Budget" (which navigates to a form to edit the budget settings) can be present. Also if the user wants to reset rollover or adjust something, that control would live here or in settings.

**Component breakdown:** We can break the UI into reusable React components: - `<BudgetProgressBar>`: a component that given `spent` and `budgetAmount` (and perhaps an optional `projected`), renders the progress bar or circle. Internally, it uses Tailwind classes for colors. For example, usage might be:

```
<BudgetProgressBar
  percent={Math.min(spent/budgetAmount*100, 100)}
  overPercent={spent/budgetAmount > 1 ? (spent/budgetAmount*100 - 100) : 0}
  status={status} />
```

This component would apply classes based on `status` prop (e.g. green vs yellow vs red backgrounds). It could also accept a type ( `"circular"` or `"bar"` ) to decide the shape. - `<ForecastToggle>`: a small component with buttons or a select for choosing forecast strategy. It would manage the state of selected strategy and call a parent handler to recalc or fetch the projected value. - `<BudgetStats>`: a simple component to display key-value pairs (spent, remaining, etc.) with proper formatting. Could be just JSX in the page as well. - `<AlertBadge>`: a component to display a colored badge or banner if the budget is in warning or overspent. For example:

```
function AlertBadge({ status }) {
  if(status === 'warning') return <span
    className="bg-yellow-100 text-yellow-800 px-2 py-1 rounded">80% Used</span>;
  if(status === 'critical') return <span
    className="bg-red-100 text-red-800 px-2 py-1 rounded">100% Reached</span>;
```

```


    if(status === 'overspent') return <span className="bg-red-100 text-red-800
px-2 py-1 rounded">Over 120%!</span>;
    return null;
}

```

This gives a quick textual alert on the UI.

All these would be composed in the **BudgetDetailPage** (for example), which might look like:

```

<div className="p-6">
  <h2 className="text-xl font-bold">Groceries  June 2025</h2>
  <div className="flex items-center my-4">
    <BudgetProgressBar type="circular" percent={90} status="warning" />
    <div className="ml-6">
      <BudgetStats spent={450} remaining={50} total={500} />
      <AlertBadge status="warning" />
    </div>
  </div>
  <ForecastToggle strategy={strategy} onChange={setStrategy} />
  <p className="mt-2 text-gray-700">
    Projected spend: <strong>${projected}</strong>
    {projected > 500 && <span className="text-red-600"> (over budget by $
    {projected-500})</span>}
    {strategy === 'linear' && <em className="text-xs text-gray-500"> at current
    pace</em>}}
  </p>
  {/* ... possibly chart or list of recent transactions ... */}
</div>

```

(Note: The above JSX is for illustration; actual code may differ. Tailwind classes like `bg-red-100 text-red-800` etc., provide the colored backgrounds for alerts.)

This per-category view allows the user to drill down and see all relevant info to that budget.

## All-Budgets Summary Dashboard

This is the main overview where the user sees a list of all their budgets and their statuses at a glance. The design should prioritize clarity so the user can identify any problem areas quickly. Features of this dashboard:

- **List or Grid of Budget Cards:** Each budget category gets a card (or a row) displaying:
  - Category name (and maybe an icon if categories have icons, e.g. a grocery cart for Groceries).
  - The budget amount and period (could be in small text, e.g. "Monthly \$500").
  - A progress indicator (bar or small circle) showing how much is used. Likely a horizontal bar fits nicely in a list row. The bar should be colored according to status (green/yellow/red).

- A percentage or amount label on the bar (e.g. "90%" or "\$450/\$500").
- Possibly the remaining amount text ("50 left") or overspent amount.
- Maybe the current status word or icon (a ●, ●, dot indicator).
- **Sorting/Ordering:** We might order the list such that overspent or critical budgets show up top, to grab attention. Alternatively, allow the user to sort or group (e.g. by personal vs business, or by status).
- **Summary Totals (optional):** If relevant, an aggregate summary can be shown at top: e.g. total budgets vs total spending. However, since budgets are category-specific, summing them might not be meaningful unless we have an overall spending limit. We can skip a total, or perhaps show "X of Y budgets are over limit" as a quick summary.

We will implement each budget entry as a React component, say `<BudgetCard>`:

```
function BudgetCard({ categoryName, period, amount, spent, status }) {
  const percentage = Math.min((spent/amount)*100, 100);
  const barColor = status === 'overspent' || status === 'critical' ? 'bg-red-500'
    : status === 'warning' ? 'bg-yellow-500'
    : 'bg-green-500';

  return (
    <div className="flex items-center justify-between p-4 border-b">
      <div>
        <h3 className="font-medium">{categoryName}</h3>
        <p className="text-sm text-gray-600">{period} budget: $
{amount.toFixed(2)}</p>
      </div>
      <div className="w-1/2 mx-4">
        <div className="w-full bg-gray-200 rounded h-3">
          <div className={`h-3 rounded ${barColor}`} style={{ width: `${percentage}%` }}></div>
        </div>
        <div className="text-sm font-semibold">
          {spent.toFixed(2)} / {amount.toFixed(2)}
        </div>
      </div>
    </div>
  );
}
```

Each `BudgetCard` will display the bar and numbers. The color of the filled part of the bar is set by the budget's status (red if critical/overspent, yellow if warning, etc.). We also show the raw numbers on the right for precision.

We can enhance the card by, for example: - Changing the text color of the amount if overspent (maybe red text if spent > amount). - Adding a small icon or badge if overspent (like a "!"). - If rollover is on and there's a carryover, we might indicate "(+\$100 carried over)" somewhere in small text.



The All-Budgets page (e.g. `/budgets`) will map through the user's budgets and render a `BudgetCard` for each. It might look like:

```
<div className="container mx-auto my-6">
  <h2 className="text-2xl font-bold mb-4">Your Budgets</h2>
  {budgets.map(budget => (
    <BudgetCard
      key={budget.id}
      categoryName={budget.categoryName}
      period={budget.period.charAt(0).toUpperCase()+budget.period.slice(1)}
      amount={budget.amount}
      spent={budget.currentSnapshot.spent}
      status={budget.currentSnapshot.status}
    />
  ))}
</div>
```

This will produce a list. We use `budget.currentSnapshot` which would have the latest `spent` and `status` (this data can come from the `budget_snapshots` table via an API call the front-end makes when loading the dashboard).

We should also implement clicking on a `BudgetCard` to navigate to the detailed per-category view (e.g. on click of the card, `router.push('/budgets/food')` or similar).

**Responsiveness & Tailwind:** Using Tailwind utility classes ensures the dashboard looks good on different screen sizes. For instance, on mobile the `BudgetCard` might stack its elements vertically. We can use Tailwind's responsive prefixes (e.g. `md:flex` for layout). The provided code is already flexible since it uses flexbox. We might adjust text sizes for small screens.

**Styling considerations:** - Use consistent spacing and padding (`p-4`, `mb-4` etc.) to make the UI clean. - Use Tailwind color shades appropriately (e.g. `red-500` for bars, and maybe `red-700` for text on light background, etc.). - Ensure sufficient contrast (the progress bar on gray background etc., which is fine in the above snippet). - Icons: if we want a small dot indicator, Tailwind doesn't have a specific component but we can use a `<span className="inline-block w-3 h-3 rounded-full bg-red-500">` as a red dot, for example.

By following these component specs, a developer can implement the UI without needing a separate mockup image. The pseudocode/JSX above serves as a blueprint of the structure and Tailwind classes to use.

---

## References:

- Budget rollover concept (carrying over unspent or overspent amounts) as described by Lunch Money and Copilot budgeting apps [1](#) [2](#).

- User request for proactive budget threshold alerts at 80% usage <sup>4</sup> , highlighting the importance of early warnings.
- Common threshold alert levels (80%, 100%, 110% etc.) and forecast alerts for budgets <sup>5</sup> <sup>12</sup> .
- Research on spending pace indicators showing green/yellow/red zones to encourage users to adjust behavior <sup>7</sup> .
- Machine learning usage for predicting expenses based on past spending patterns <sup>9</sup> , offering improved forecasting accuracy over simple rule-based methods <sup>10</sup> .

---

<sup>1</sup> <sup>3</sup> **What Is a Budget Rollover?**

<https://lunchmoney.app/blog/what-is-a-budget-rollover>

<sup>2</sup> **Budget Rollovers | Copilot Help Center**

<https://help.copilot.money/en/articles/3790828-budget-rollovers>

<sup>4</sup> <sup>6</sup> <sup>8</sup> **Budget Thresholds with Notifications : r/MonarchMoney**

[https://www.reddit.com/r/MonarchMoney/comments/1coonxf/budget\\_thresholds\\_with\\_notifications/](https://www.reddit.com/r/MonarchMoney/comments/1coonxf/budget_thresholds_with_notifications/)

<sup>5</sup> <sup>12</sup> <sup>13</sup> **How to Set Up Cost Alerts and Budgets in Azure - Inventive HQ**

<https://inventivehq.com/knowledge-base/microsoft-azure/%E2%9C%85-how-to-set-up-cost-alerts-and-budgets-in-azure/>

<sup>7</sup> **US20170287064A1 - Methods and apparatus for promoting financial behavioral change - Google Patents**

<https://patents.google.com/patent/US20170287064A1/en>

<sup>9</sup> **Creating a Smart Personal Finance Tracker with Python and Machine Learning | by RandomResearchAI | Medium**

<https://randomresearchai.medium.com/creating-a-smart-personal-finance-tracker-with-python-and-machine-learning-65050e916089>

<sup>10</sup> <sup>11</sup> **irjmets.com**

[https://www.irjmets.com/uploadedfiles/paper//issue\\_3\\_march\\_2025/69060/final/fin\\_irjmets1741847068.pdf](https://www.irjmets.com/uploadedfiles/paper//issue_3_march_2025/69060/final/fin_irjmets1741847068.pdf)