ChatGPT

Unified Revenue and Expenses Engine: Implementation and Design Guide

Introduction

A unified revenue and expenses engine is a financial analytics system that consolidates all raw financial data – sales transactions, fees, refunds, chargebacks, payouts, subscriptions, and product data – into a single source of truth. This guide outlines a comprehensive design and implementation strategy for such an engine. It covers data modeling (including tagging and multi-currency handling), the core revenue and expense calculation logic, reconciliation processes, projection of summary data (using CQRS principles), API design, governance features, and integration considerations. We will also provide pseudocode for key aggregations, example data schemas in both NoSQL (Firestore JSON) and SQL, discuss technology trade-offs (Firestore vs. BigQuery vs. Postgres), compare relevant tools (like dbt, Airbyte, Stripe SDK), and include illustrative diagrams.

Data Model and Normalization

Raw Financial Entities: At the heart of the system are raw financial records from various sources. These include individual transactions (sales invoices or orders), payment processor fees, customer refunds or chargebacks, payout records from payment gateways (e.g. Stripe, Mercado Pago), subscription records, and product definitions. Each record should carry essential attributes like date/time, amount, currency, links to related entities (e.g. a fee links to the sale it applies to, a payout links to the transactions it covers), and metadata such as payment source or platform. Maintaining a unified schema for these heterogeneous inputs is crucial – often achieved by an **event** or **ledger** model where every financial event (sale, refund, fee, etc.) is a transaction entry with a type and references.

Tagging Model with Inheritance: To enable flexible reporting and categorization, the engine should support tagging financial records with categories and labels. For example, a transaction might be tagged with a product category (“Software” vs “Hardware”), a revenue source (“Online Store” vs “Marketplace”), or an expense category (“Marketing”, “SaaS Subscription”, etc.). A tag model with inheritance means tags can be applied at different levels of hierarchy and propagate down unless overridden. For instance, a *Product* entity could have a default tag (like “Category: Software”) that is inherited by all sales of that product, while a specific sale can still be overridden with a different tag if needed (user override). This inheritance allows broad rules (e.g. tag all transactions from Vendor X as “Office Supplies”) while still permitting exceptions on individual records. If a parent tag is changed, child records should update unless they have an explicit override. The system should log any user overrides to tags for auditability (addressed further under **Governance**).

Multi-Currency Normalization: Companies operating in multiple currencies need a consistent way to consolidate financials. The engine should convert all amounts to a base currency for reporting, using defined valuation date rules. Typically, the **transaction date** or a period-end date is used as the FX valuation date. For each transaction in a foreign currency, the exchange rate for the chosen date is applied

to normalize into the base currency [1]. It's important to source reliable FX rates (via an API or financial data feed) and to handle edge cases – for example, a transaction occurring near midnight might have ambiguity on which day's rate to use [1]. To maintain accuracy, define clear rules (e.g. use the prior day's closing rate vs. same-day rate). The engine should also accommodate **late FX adjustments**: if actual settlement occurs later at a different rate, the difference might be recorded as a small FX gain/loss entry. When reconciling multi-currency amounts, implement tolerance thresholds to allow minor discrepancies due to rounding or rate timing. For example, you might permit a difference of up to $0.10 or 0.1% in any currency to account for rounding in conversions [2] [3] – this prevents chasing trivial differences that are not material to financial reports.

Revenue Engine

The revenue engine is responsible for computing key sales metrics and aggregating income data across various dimensions.

- **Gross vs. Net Revenue:** Gross revenue is the total sales amount before any deductions. Net revenue is calculated by subtracting related costs such as payment processor fees, refunds, and chargebacks from the gross revenue. The engine should accumulate both. For example, if $10,000 in sales occurred in a day with $500 in refunds and $300 in fees, gross revenue = $10,000 and net revenue = $10,000 – $500 – $300 = $9,200.

- **Units Sold and AOV:** In addition to monetary values, track the number of units sold (quantity of products/services) and the number of orders. This enables computing the Average Order Value (AOV = gross revenue / number of orders) as a measure of transaction size. If needed, also track metrics like average selling price per unit, conversion rates, etc., but AOV and unit counts are the primary volume indicators for revenue.

- **Time Granularity:** The engine should support aggregating these metrics by different time grains – daily, weekly, and monthly at minimum. This allows generating daily reports, as well as roll-ups by week or month for trend analysis. The aggregation logic should be consistent across grains (e.g. monthly being an aggregation of the daily records within that month).

- **Breakdowns by Dimension:** Revenue can be broken down by product, product category, sales channel/source, region, or any tag. The engine's design should make it easy to produce segmented revenue reports. For example, one should be able to see revenue per product line or compare revenue from the website vs. a marketplace. This can be handled by grouping on those dimensions in queries or by maintaining separate summary tables keyed by dimension. Tag inheritance helps here (if all products under "Category X" are tagged, the system can sum all revenue with that tag to get Category X revenue).

Payout Reconciliation

A critical part of revenue operations is ensuring that the cash received matches the sales recorded – this is handled through payout reconciliation for payment platforms (Stripe, Mercado Pago, PayPal, etc.). These platforms typically aggregate transactions (minus fees and refunds) into periodic payouts that appear as deposits in your bank account. Our engine should automate reconciliation of these payouts to the underlying transactions:

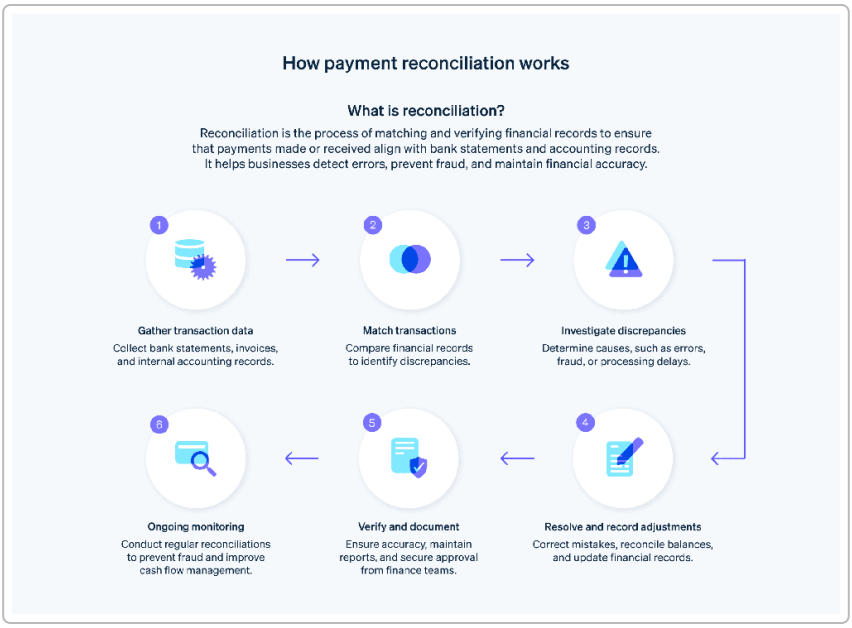


Illustration: The payout reconciliation process involves gathering transaction records, matching them to bank deposits, investigating discrepancies (e.g. fees or FX differences), resolving issues with adjustments, and ongoing monitoring of any unmatched items. Each numbered step corresponds to part of the workflow: (1) Gather all transaction data (sales, refunds, fees, etc. from the source system) and the payout records; (2) Match transactions to payouts by comparing totals and dates; (3) Investigate any discrepancies (such as missing transactions, timing issues, or errors); (4) Resolve differences by applying adjustments or corrections; (5) Verify and document the reconciliation results; (6) Monitor regularly to catch new discrepancies early.

Reconciliation Steps: To reconcile, the engine should first gather all relevant data – this means retrieving the detailed transaction report from the payment provider for the payout period, and the corresponding deposit entry from the bank statement [4]. Next, it will **match transactions** to the payout: sum up all sales in that payout minus refunds and fees, and compare that net sum to the deposit amount [5]. If everything is in sync, the payout is fully matched. Often, however, there are discrepancies due to timing (e.g. a transaction that settled a day later), fees, or currency conversions. The engine should **identify discrepancies** by cross-checking each component [6] – for example, does the total fee amount in Stripe's report equal the difference between gross and net? Are all refunds accounted for? In multi-currency scenarios, differences might arise if Stripe converted funds to the bank currency at slightly different rates; here the system may use a tolerance (as discussed) to auto-clear small variances. Any difference beyond tolerance should be flagged for investigation [7].

FX Adjustments and Tolerances: If a payout involves converting currencies (e.g. Mercado Pago depositing BRL for sales made in USD), the engine must factor in the FX rate used by the provider. Often the provider's report will include both the original currency amount and the converted amount. The reconciliation logic can perform an "FX check" – re-convert the original amounts using known rates to verify the provider's conversion [8]. A minor difference might be acceptable; for instance, allowing a 0.1% mismatch as tolerance [3]. If the difference exceeds tolerance, it could indicate an error in the rate or an unrecorded fee, prompting further review. Similarly, a "fee check" ensures the fees match the provider's fee schedule [9] – if not, either the fee was recorded incorrectly or perhaps an extra charge (like a fixed per-payout fee) needs to be applied.

Automation and Matching Logic: The system can implement a matching algorithm that tries to pair each bank deposit with the corresponding payout record from Stripe/Mercado Pago. One approach is multi-line matching: e.g., if the bank shows one deposit of $5,000, the engine may need to combine multiple smaller payouts (or vice versa) to explain that deposit. The engine should support one-to-one, one-to-many, or many-to-one matches if the payout frequencies differ from bank posting (though typically, each payout corresponds to one deposit). When a match is found within tolerance, it's marked reconciled with a status. If not, the payout or the deposit remains open (unreconciled) for manual investigation. All reconciliation actions should be logged, and the system could provide an interface to manually match or adjust entries if needed.

Output: The result of payout reconciliation is typically a report or read model (see **Projections** below) that lists each payout and indicates its status: Matched in full (no discrepancy), Matched with small adjustment (and what adjustment was made), or Unmatched (with amount differences). This gives finance teams an at-a-glance view of whether all platform revenues have actually been received in the bank. Regular reconciliation (e.g. monthly) helps detect fraud or errors early [10] and ensures accurate financial statements [11].

Subscription Metrics (MRR & Cohorts)

Many businesses have recurring revenue streams (subscriptions), so the engine should also handle SaaS metrics: **Monthly Recurring Revenue (MRR)** and subscription retention/churn.

- **MRR Calculation:** MRR is the normalized monthly revenue from all active subscriptions. The engine can compute MRR by summing the monthly subscription price of every active customer, or equivalently multiplying the number of active subscribers by the average revenue per user [12]. For accuracy, it should account for partial periods and proration if subscriptions start mid-month or have upgrades/downgrades. It's helpful to break MRR into components as well: new MRR from new customers, expansion MRR from upgrades, churned MRR from cancellations, reactivation MRR, etc. [13] [14]. These components allow computing **Net New MRR** each month (new + expansion minus churned) [14]. For example, if in September a company added \$2,000 MRR in new subscriptions, lost \$500 MRR from cancellations, and gained \$300 MRR from existing customers upgrading, the net MRR change = +\$1,800.

- **Retention and Cohort Analysis:** Cohort analysis groups customers by their start month and tracks metrics over time. The engine should support cohort-based retention tables for subscriptions. Each cohort (e.g. customers who subscribed in May 2021) is tracked to see what percentage remain subscribed (or what MRR remains) after 1 month, 2 months, 3 months, etc [15]. The output can be a cohort matrix where the first column is the cohort start month and subsequent columns show retention rates or remaining MRR by month. This helps identify how quickly subscribers churn. For instance, an analysis might show that of all customers who joined in May 2021, 100% were still active after 1 month, 90.9% after 4 months, and only 63.6% were active by month 12 [16]. Such insight highlights long-term retention and can be used to compute metrics like average customer lifetime or churn curves. The engine might compute these by joining subscription data across months or by capturing cancellations and computing survival rates. MRR cohorts are similar: instead of customer counts, track the MRR value retained from each cohort over time (this accounts for upgrades/downgrades in revenue, not just customer logos). Both views are useful – customer retention shows

logo churn, MRR retention shows revenue churn, which can differ if surviving customers change plans.

- **Cohort Outputs:** The engine can produce a JSON or table of cohort data ready for the UI. For example, a JSON might have an entry: `{cohort: "2021-05", month_0_MRR: $X, month_1_retention: 100%, month_12_retention: 63.6%}` etc. Alternatively, the engine can just produce time-indexed retention percentages and the front-end will format into a matrix or chart.

Implementing these subscription metrics likely involves scanning the subscription records month by month. It may be useful to create an intermediate model, e.g., a table of monthly snapshots of active subscriptions and their MRR, which can then be used to derive MRR and retention over time via SQL queries or code.

Expenses Engine

The expenses engine aggregates and analyzes outgoing money flows with similar rigor, ensuring the company's costs are categorized and monitored.

- **Aggregation by Category and Vendor:** Just as revenue is broken down by product or source, expenses should be rolled up by meaningful categories. Categories could include predefined accounting categories (Marketing, Salaries, Software, Travel, etc.) or custom tags. The engine should sum expenses by these categories for any given period. It should also support aggregation by merchant/vendor or payee. For example, a report could show total spend at AWS vs. Google Cloud over the quarter, or total "Marketing" expenses per month, etc. Tagging comes into play heavily here: each expense transaction (e.g. a bank transaction or bill) can carry a tag for its category, and possibly multiple tags (e.g. "Recurring" and "Software" for a monthly SaaS subscription payment to an email service). The inheritance model can apply here as well: e.g. tagging a vendor as "Utilities" so all bills from that vendor default to Utilities category unless overridden.

- **Account and Payment Method Breakdown:** It can also be useful to break expenses down by the account or card used (if the company has multiple bank accounts or credit cards). The engine could aggregate by account to help reconcile bank statements or to see which accounts are paying which expenses.

- **Recurring Expense Detection:** A standout feature for expense analytics is identifying recurring expenses. Recurring expenses are those that happen on a regular interval (typically monthly or annually) for similar amounts – for instance, subscriptions to software, rent payments, or service contracts. The engine can automatically flag transactions that appear to repeat. One approach is to look for the same vendor or description occurring at least 3 times with roughly the same amount and a regular interval (e.g. ~30 days apart). For example, three payments to "Adobe Cloud" around the 1st of Jan, Feb, Mar for \$50 suggests a monthly recurring expense. Tagging these as "recurring" helps in budgeting and in anomaly detection. The system could maintain a list of identified recurring expenses and their schedule, which also aids cash flow forecasting. Users could be allowed to confirm or adjust these (for instance, mark an expense as non-recurring if incorrectly flagged).

- **Outlier Detection:** To control costs, it's important to spot unusual spikes or drops in expenses. The engine should apply basic statistical anomaly detection on expense data. A simple method is using a

z-score (number of standard deviations from the mean) on time series of expenses by category or vendor. For example, if typical monthly spend on “Office Supplies” is \$1,000 with small variance, but this month it’s \$5,000, the z-score would be high and the system flags it as an outlier for review. Another more robust approach is using STL (Seasonal-Trend Decomposition) on the expense time series to remove seasonal patterns and identify anomalies in the residual. Seasonal patterns might include higher utility bills every winter or annual insurance payments – STL can model those, and anything not explained by the trend or seasonality is an anomaly. These techniques help filter out false alarms and focus attention on true irregularities. The expenses engine could present alerts like “Marketing spend in Week 36 is 3σ above average – investigate possible causes.” As an example from industry, Tink’s financial platform uses outlier detection to filter out transactions with unusually large amounts (which could skew analyses) [17] – similarly, our engine might exclude one-off huge expenses when analyzing typical monthly costs, or at least flag them.

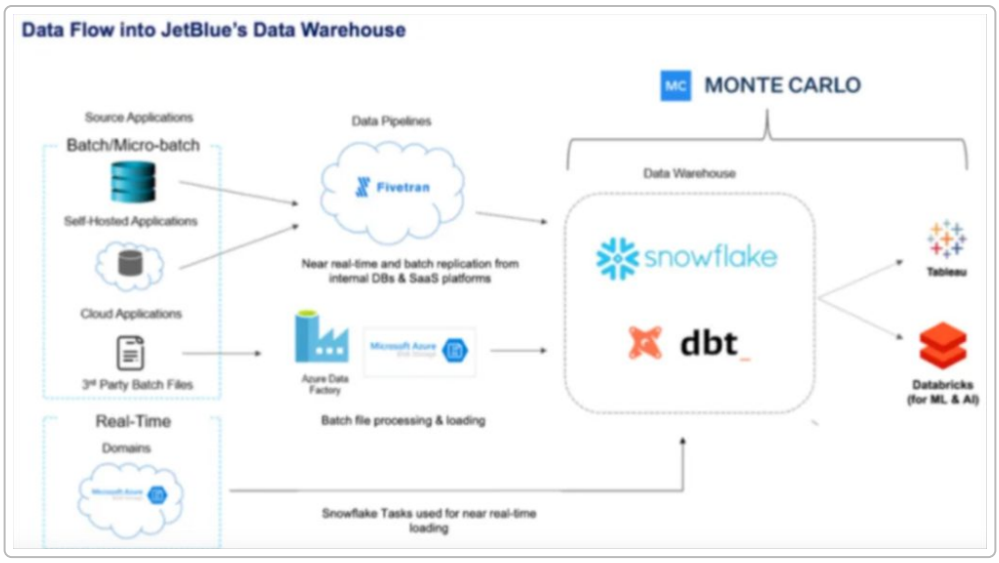
- **Net vs. Gross Expenses:** Sometimes expenses are refunded or rebated (e.g. you purchase something and later get a refund, or a credit card cashback). The engine should account for these by computing gross expenses (total outflow) and net expenses (after any refunds/credits). For instance, if an employee bought a \$500 flight ticket (expense) but later the trip was canceled and \$500 was refunded to the card, the gross expense is \$500 but net expense is \$0 since it was refunded. It may be useful to mark such offsets in the data (perhaps linking the refund transaction to the original expense). In accounting terms, this helps to avoid overstating expenses and also aids in auditing (matching refunds to original spends). The expense engine can automatically match refunds to expenses by vendor/amount/date or use reference IDs if available (like a return authorization number, etc.). Net expense calculations should be reflected in the summaries so that, for example, the monthly expense summary shows net values unless gross breakdown is specifically needed.

Expense Anomaly Handling: If an outlier is detected or a recurring expense changes significantly (say a usual \$200/month bill jumps to \$500 one month), the system should flag it. This could trigger a workflow (maybe an alert or moving the item to an “Exceptions” report for finance to approve or annotate). These governance aspects tie into the next sections.

Data Ingestion and Processing Pipeline

Before diving into how the data is aggregated and exposed, it’s worth outlining how data gets into the system and flows through it. A unified engine often pulls data from numerous sources: payment processors (Stripe, PayPal, Mercado Pago APIs), banking systems (through file imports or bank APIs), subscription management systems, e-commerce databases, etc. Designing a robust **data pipeline** is crucial to feed the engine with up-to-date and accurate data.

Modern best practices suggest using ELT pipelines (Extract-Load-Transform) with high-quality connectors and a central data warehouse. For instance, an open-source tool like **Airbyte** or a service like **Fivetran** can handle extraction from external APIs and loading into a database. Real-world teams commonly load raw data into a cloud warehouse (e.g. BigQuery or Snowflake) and then run transformations to produce the aggregated models [18] [19]. This approach decouples data retrieval from processing. The advantage is connectors handle API quirks, and the heavy lifting of aggregation can happen in SQL on a scalable warehouse.



Example Data Pipeline Architecture (inspired by JetBlue's implementation [18]): Source applications (payment processors, SaaS apps, internal DBs, etc.) send data in batch or real-time. An ingestion service (e.g. Fivetran or Airbyte) replicates raw transactions into a cloud data warehouse (like Snowflake or BigQuery). Transformation tools (like dbt) then convert raw tables into cleaned, aggregated tables (e.g. daily revenue, monthly expenses). The transformed data is used by analytics and BI tools (or in our case, by the unified engine's APIs and a React/Next.js UI). Monitoring and observability tools (e.g. Monte Carlo, not shown) can oversee the pipeline's health.

In our design, since we target a Next.js + TypeScript stack for APIs and UI, we could incorporate the data processing in the application or use a warehouse in the backend. There are two main patterns:

- **All-in-App Processing:** Here, the Next.js server (Node.js environment) would directly call source APIs (e.g. Stripe SDK to fetch transactions) on a schedule or via webhooks, store raw data in a database (Cloud Firestore or Postgres), and compute aggregations on the fly or in background jobs. This might leverage serverless functions or cron jobs to periodically recompute summaries. It simplifies architecture (no separate warehouse), but can become complex as data grows or if transformations get heavy.

- **Warehouse + Transform (ELT) Approach:** In this pattern, the engine uses an external database to do the heavy data crunching. For example, use BigQuery (if on GCP) to store raw data tables and define SQL views or use a tool like **dbt** to create the summary tables. The Next.js API layer would then simply query these precomputed tables (either via an API or using a client library) to serve the UI. Using a warehouse can handle large volumes easily and benefit from SQL optimization. The downside is additional infrastructure and possibly higher latencies for small queries if not cached.

A hybrid approach can also work: use Firestore (NoSQL) for quick real-time lookups and minimal data (perhaps storing the latest daily summaries or any user override data), and use BigQuery for heavy historical analysis or backfills.

Regardless of approach, **idempotency and automation** in the pipeline are vital. If using Airbyte/Fivetran to load data, they typically ensure incremental loads without duplication. If building custom, ensure that ingest routines either upsert data by a unique ID or ignore duplicates to avoid double-counting

transactions. Similarly, transformation queries should be idempotent – meaning running them twice yields the same result, not double-counting data [20] [21]. This can be achieved by always recomputing results from source data (rather than adding to existing totals) or by using **MERGE/UPSERT** logic in SQL.

For example, an idempotent daily revenue calculation might first delete any existing summary for that day and then insert a fresh aggregation, or use a single SQL `MERGE` statement to upsert the new totals. The goal is that if a job fails halfway or runs twice, the final data isn't corrupted [22] [21]. This design also simplifies **backfills** – if we onboard historical data or fix an error, we can re-run the aggregation for past dates freely. In an idempotent pipeline, updating business logic and recomputing is straightforward since each run produces the same correct state [23] (if logic changes, you rebuild the derived tables without having to manually clean duplicates, which you'd have to do in a non-idempotent "append-only" pipeline). Our engine will leverage this principle heavily in the **Projections** stage.

Summary Projections and CQRS Read Models

To serve data efficiently to the frontend and APIs, the system will maintain **projection tables (or documents)** that aggregate the raw transactions into summary views. This follows the CQRS (Command Query Responsibility Segregation) pattern: the raw data is the write model (detailed transactions, highly normalized), and the read models are pre-computed denormalized views optimized for queries [24] [25].

Key read models (projections) to implement:

- `revenue_summary_day` – aggregated revenue metrics per day. Each record (or document) represents one day of revenue, containing fields like total gross revenue, net revenue, number of orders, units sold, and possibly breakdown subfields for each product or source for that day. This model allows quickly plotting a daily revenue graph or retrieving yesterday's KPIs. A similar model `revenue_summary_month` can store monthly aggregates (or it can be computed on the fly by summing the daily, depending on volume).

- `expense_summary_day` – analogous to revenue summary but for expenses. Each entry has total expenses for the day (gross and net of refunds), possibly broken down by category or account. And similarly `expense_summary_month` for monthly totals.

- `payout_reconciliation` – a table or collection that tracks each payout (and potentially each bank deposit) with reconciliation info. For example, each entry might include: payout ID, date, amount, status (unmatched, matched, or matched_with_diff), the matched bank transaction ID, and the list of constituent transactions or a reference to them. This projection is updated whenever new transactions or payouts come in or when a reconciliation is performed. It allows an API or UI to query "show me all payouts that aren't reconciled yet" or "details for payout #1234".

- `mrr_summary_month` (optional) – could hold MRR and subscriber counts per month, and maybe churn rates, etc., to assist in quickly drawing MRR trending charts or calculating month-over-month growth without scanning all subscriptions each time.

- **Cohort tables** – e.g. a structure where each record is a combination of cohort and period with retention info. For instance, fields: {cohort_start_date, months_from_cohort, cohort_size,

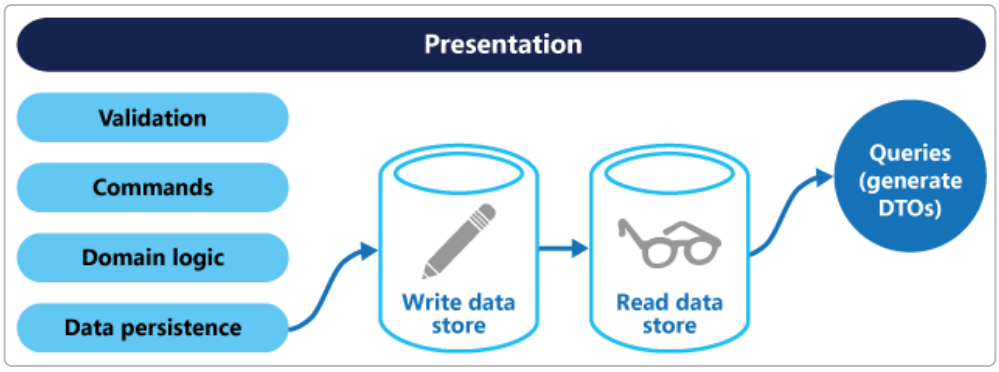
remaining_customers, remaining_MRR}. This could also be stored in a nested form (cohort as a document with an array of monthly retention numbers). Having this as a read model means the heavy computation of cohorts (which might require scanning all subscription histories) can be done periodically and stored, rather than recalculated on every request.

The engine's **write side** will continuously ingest or accept new records (e.g. a new sale comes in from Stripe webhook, or a new expense transaction is imported). When these events occur, the system triggers updates to the read models. This can be done in two ways:

1. **On-the-fly calculation:** e.g. when a new transaction arrives, increment the corresponding daily summary counters. For instance, add the transaction's amount to that day's gross and net revenue, increment the order count, etc. This is immediate and keeps summaries up to date in near real-time. If implementing in Firestore, a cloud function or server logic can do a transaction to update the summary document upon each new write. Care must be taken to handle race conditions or multiple events for the same day (Firestore transactions can ensure atomicity in increments). This approach is event-driven and aligns well with CQRS event handlers updating projections.

2. **Batch recomputation:** alternatively, update the projections in batches (e.g. recompute yesterday's totals each night, or use a scheduled job that processes any new transactions since last update). This is simpler in terms of consistency (always computing from scratch for a time window ensures accuracy), but introduces slight lag.

Using CQRS, one could even separate the data stores: e.g., keep the raw transactions in Postgres (for reliability and complex querying if needed) and store the read models in a fast NoSQL store or as cached JSON. Some implementations use event sourcing: each transaction is an event in an event log, and the read models are built by consuming that event log. For our scope, a simpler approach of using database triggers or application logic to maintain the summaries is sufficient.



Architecture Diagram – CQRS with Separate Read/Write Data Stores: Commands (writes) enter the system and update the write model (e.g. raw transaction store). An event or message is then sent to update the read model (the precomputed summary in a separate store). The presentation layer (queries) only reads from the read model, ensuring fast query responses optimized as DTOs. In our context, this could mean using a normalized SQL database for transactions (write model) and a de-normalized Firestore (or Redis cache, etc.) for read models, or simply maintaining separate tables in the same database.

Idempotent Upserts & Late Arriving Data: When updating the read models, use upsert logic so that running the update multiple times yields the same result. For example, if a late-arriving transaction from last week appears (perhaps a backdated entry that was just now imported), the engine should recompute that day's summary either by subtracting the old value and adding the new, or by fully recalculating that day. A straightforward way is to have the daily summary projection be overwritten for that date (SELECT sum(...) GROUP BY date for that specific date) whenever changes happen. SQL `MERGE` or `ON CONFLICT DO UPDATE` is useful for this: the system can attempt to insert a new summary row and on conflict (if one exists) update it with the new totals. In pseudo-SQL:

```
-- Pseudocode: upsert daily revenue summary for a given date
MERGE INTO revenue_summary_day AS summary
USING (
    SELECT
      t.date AS date,
      SUM(t.amount) AS gross_revenue,
      SUM(t.amount - t.fee - t.refund) AS net_revenue,
      COUNT(DISTINCT t.order_id) AS orders,
      SUM(t.quantity) AS units
    FROM transactions t
    WHERE t.date = @target_date
    GROUP BY t.date
) AS computed
ON summary.date = computed.date
WHEN MATCHED THEN
   UPDATE SET gross_revenue = computed.gross_revenue,
              net_revenue   = computed.net_revenue,
              orders        = computed.orders,
              units         = computed.units,
              aov           = computed.gross_revenue / NULLIF(computed.orders,0)
WHEN NOT MATCHED THEN
   INSERT (date, gross_revenue, net_revenue, orders, units, aov)
   VALUES (computed.date, computed.gross_revenue, computed.net_revenue,
computed.orders, computed.units,
           computed.gross_revenue / NULLIF(computed.orders,0));
```

The above pseudocode (which could be adapted to Firestore or a MapReduce job) computes the daily totals and either updates or inserts the summary row. By doing this for each date where transactions changed, the model stays accurate. It also means if the job is rerun for the same date, the result doesn't double count – it replaces with the same values [26] [21].

Late-arriving data is handled by the fact that any date's summary can be recalculated when needed. The system might maintain a queue or log of which dates (or which payout IDs, etc.) need recomputation due to new or changed data. For example, if a refund for a last month's transaction comes in today, the engine should adjust last month's net revenue downward. This trigger could fire automatically when the refund is recorded, marking the affected period's summaries for update.

Backfilling: When initially launching or whenever a major change occurs (like re-categorizing many transactions), you may need to rebuild summaries for a large historical range. Because our transformations are idempotent and based on raw data, we can truncate the summary tables and rebuild entirely from scratch if needed – a process that might be automated or done on-demand. Ensuring the ability to backfill (without conflicting with live updates) is part of the design. One strategy is to have a backfill routine that can run in a separate environment or on the warehouse, and then swap out the old data. Another is to gradually backfill by date (so the system always has something to serve, and backfills older data behind the scenes).

In summary, the projections (read models) are the prepared, easy-to-query datasets that our APIs will serve. By maintaining them with CQRS and idempotent updates, we ensure consistency and performance in the face of ongoing data changes.

API Design for Data Access

The engine will expose a set of RESTful (or GraphQL, but we'll assume REST/JSON for now) endpoints to allow the frontend (Next.js/React UI) or external systems to retrieve the financial summaries and reconciliation info. Given we have distinct domains (revenue, expenses, payouts), we can design separate endpoints for clarity. All endpoints should support filtering by date ranges, and possibly by tags/dimensions, and return data in a format ready for charts or tables in the UI.

Key API endpoints include:

- **Revenue Summary Endpoint:** e.g. `GET /api/revenue/summary` – This could return aggregated revenue over a period. Query parameters might include: `granularity` (day, week, month), `start_date`, `end_date`, and optional filters like `product` or `source`. For example, `GET /api/revenue/summary?granularity=month&start=2025-01&end=2025-12` would return an array of monthly revenue objects for 2025. Each object might have fields: `{ period: "2025-01", gross_revenue, net_revenue, orders, units, aov }`. If breakdown by product is needed, either the endpoint can accept a `group_by=product` and return multiple series (like a nested JSON with each product as a key), or we provide a separate endpoint. Another approach is a more general query endpoint like `GET /api/revenue/by-product` which returns the breakdown by product for a given period. However, to keep things simple, one could design the summary endpoint to include breakdowns if no specific filter is given. For instance, if the request has `breakdown=product`, the response could nest the data by product. The DTOs (Data Transfer Objects) should be **chart-ready**, meaning if the frontend expects an array of points or a structure keyed by labels, the API should format data accordingly to minimize frontend processing. An example response snippet might be:

```
{
  "granularity": "month",
  "start": "2025-01",
  "end": "2025-03",
  "currency": "USD",
  "totals": [
    { "period": "2025-01", "gross": 12000, "net": 11500, "orders": 300,
```

```
"units": 320, "aov": 40.0 },
    { "period": "2025-02", "gross": 13000, "net": 12500, "orders": 310,
"units": 330, "aov": 41.9 },
    { "period": "2025-03", "gross": 14000, "net": 13500, "orders": 320,
"units": 340, "aov": 43.8 }
  ],
  "breakdown_by_product": {
    "Product A": [ { "period": "2025-01", "gross": ..., ...}, ... ],
    "Product B": [ { ... } ]
  }
}
```

In the above, `breakdown_by_product` would appear only if requested. The API design should weigh whether to include breakdowns in the same payload or have separate endpoints; including them can make the payload large if many products, so often a separate call per breakdown dimension is fine (e.g. the UI calls summary API for totals and also calls another for product breakdown if needed).

- **Expense Summary Endpoint:** e.g. `GET /api/expenses/summary`. Similar to revenue, it would provide totals by day/week/month for expenses, with filters for category, account, or tag. For instance, a filter could be `category=Marketing` to get only marketing expenses. Or `merchant=AWS` to get expenses for a specific vendor. If no filter, it returns all expenses aggregated. Breakdown could be by category by default (since companies often want to see a pie chart of expenses by category). The DTO might look like: `{ period: "2025-Q1", total_expenses: X, byCategory: { "Marketing": Y, "Salaries": Z, ... } }`. For charting, if showing a time series of expenses split by category, the API might need to return a structure where each category is a series. This can be achieved by similar breakdown parameter approach.

- **Payout Reconciliation Endpoint:** e.g. `GET /api/payouts/reconciliation`. This would list payouts and their statuses. Likely filters could be by status (unmatched only, or all) and by date range (e.g. payouts in last 30 days). The response might be a list of payouts:

```
[
  { "payout_id": "po_12345", "date": "2025-09-01", "amount": 5000,
"currency": "USD", "status": "matched", "difference": 0.00 },
  { "payout_id": "po_12346", "date": "2025-09-02", "amount": 3000,
"currency": "USD", "status": "unmatched", "difference": -50.00 }
]
```

For each payout, the UI might then query a detail endpoint like `GET /api/payouts/reconciliation/{payout_id}` to get specifics: e.g. a breakdown of which transactions are in that payout and which bank deposit matched. The details could include arrays of transaction IDs or even full transaction objects, and any notes on discrepancies (e.g. “Missing $50 – likely currency conversion difference”). Providing a separate detail endpoint keeps the list call lightweight. The

match status could be an enum (Matched, Unmatched, Partial) or simply a boolean plus a difference amount. If there are many payouts, pagination would be needed.

- **Subscription Metrics Endpoints:** Perhaps `GET /api/subscriptions/mrr` for MRR over time and `GET /api/subscriptions/cohorts` for cohort data. The MRR endpoint would return a time series of MRR (with breakdowns by new/expansion/churn if desired), while the cohorts endpoint might return a matrix or list of cohorts. For example, the cohorts endpoint could return something like:

```
[
  { "cohort": "2021-05", "month0_customers": 11, "month0_MRR": 1100,
"month1_retention": 100.0, "month3_retention": 90.9, "month12_retention":
63.6 },
  { "cohort": "2021-06", "month0_customers": 15, ... }
]
```

(Retention could be given in percentage or as remaining count/MRR, and the frontend can convert to percentage.) Alternatively, the API might provide a more nested structure: each cohort with an array of monthly points. The decision depends on ease of use with the charting library on the frontend. Typically, for cohort tables, sending the raw matrix data is fine.

All endpoints should be secured (since financial data is sensitive) – in a Next.js app, this could be done via API route middleware checking authentication. Since the question focuses on implementation design, we assume the user management and auth is handled elsewhere.

The **DTO design** is crucial: we want to minimize the amount of transformation the frontend needs to do. For example, if the frontend needs to draw a line chart of daily revenue, the API should ideally already give an array of {date, revenue} objects or parallel arrays for X and Y axis. If using a library like Chart.js or similar, we tailor the JSON accordingly.

Finally, consider adding **export** capabilities: e.g. `GET /api/revenue/summary.csv` could return the data in CSV format for download (for users who want to load into Excel). This is not required but often useful in financial systems.

Governance and Data Management

Financial data systems require strong governance to ensure accuracy, auditability, and traceability. Our unified engine should incorporate the following governance and management features:

- **Recalculation Triggers:** As data changes, we need to trigger recalculations of derived numbers. Some triggers are automatic – e.g., when a new transaction is ingested, it should trigger an update to today's summary. Others might be manual or conditional – e.g., a user might request a "recalculate" if they suspect data is off or after changing some categorization rules. We can provide an administrative function to recompute summaries for a given date range or to force reconciliation re-attempt. Ensuring idempotent, on-demand recalculation is possible was a key design goal (to

allow backfills and late data handling). For instance, if an error is found in the source data that affected last quarter's numbers, after correcting the data, an admin could trigger a re-run for that quarter to update all summaries. The system might also schedule periodic full recomputation (like nightly rebuild) as a safety net to correct any drift.

- **Audit Logging for Overrides and Manual Adjustments:** Whenever a user intervenes – for example, manually matching a payout to a deposit, overriding a tag on a transaction, or editing a transaction's details – those changes should be logged. The log should record what was changed, the old value, the new value, who made the change, and a timestamp. This is important both for compliance (especially if this system feeds accounting books; auditors will want to see a trail of changes) and for debugging (if a number looks wrong, the log might reveal that someone recategorized a huge expense into a different category, etc.). For tag inheritance specifically, if a user overrides an inherited tag on a particular transaction, that override should be stored such that it's clear it wasn't the default. Ideally, the system could maintain both the original classification and the override, so one can trace why an aggregate is calculated the way it is ("$500 was counted under Marketing because user X re-tagged transaction #123 from Office Supplies to Marketing on 2025-09-10"). Depending on scale, these logs could be kept in a separate collection/table (e.g. an "audit_log" table) or as part of each record's history (some databases like Firestore allow keeping a history array on the doc, but that can grow messy; a separate log is cleaner).

- **Data Lineage and Traceability:** The engine should enable data lineage tracking – meaning for any aggregated number, you can trace back to the raw inputs. This is vital in financial contexts to answer "what makes up this number?". For example, if the daily revenue summary says \$50,000 gross on 2025-09-01, the system should be able to produce the list of transactions that sum up to that \$50,000 [27]. In practice, this could be done by simply querying the raw transactions by date, but if any filtering or tagging is involved (e.g. "Marketing expenses in Q1"), the system should use the same criteria to fetch underlying records. One way to facilitate this is to store references in the summary projections. For instance, a summary document might hold an array of transaction IDs or a hash of them. However, listing potentially thousands of IDs in a summary record isn't ideal. A better approach is to provide APIs or queries to get the detail on demand: e.g., an API `GET /api/revenue/summary/2025-09-01/details` could fetch the raw transactions for that day (or maybe top N transactions by size, etc., with a link to download full list if needed). For reconciliation entries, the payout record could directly list the transaction IDs included, making it straightforward to see which ones were matched. Maintaining lineage might also involve ensuring that each transaction has a unique identifier and that ID is stored wherever the transaction's data is used (like in a join or in the payout's records). If we use a data warehouse and SQL, lineage is more about having good documentation of which source tables feed which derived tables (tools like dbt come in handy here as they can show a DAG of models).

- **Handling Corrections:** In finance, sometimes past data needs correction (e.g., a mis-categorized expense, or a transaction initially missed). The engine should handle late corrections gracefully. For categorization changes, the effect is typically just that aggregated totals by category change (net total expense might remain same but move from one category to another). In such cases, a recalculation trigger for those categories/time periods should occur, as mentioned. For added or removed transactions (late-arriving or voided records), the summaries and possibly reconciliations need updating. We must also consider if the system itself allows creating "adjustment entries." For example, accounting often does adjusting journal entries. If this engine will interface with

accounting, perhaps we allow an admin to input an adjustment (like "+ \$100 to Marketing on 2025-09-30 to correct a missing item") which is treated like a special transaction (with a tag like "adjustment") that then flows through the system. All such adjustments should also be clearly flagged and logged.

- **Security and Access Control:** Though not explicitly asked, any financial engine should enforce that only authorized personnel can view or modify sensitive data. For instance, expenses might be more restricted (if they contain salary info, etc.). Since this is a design guide, one would mention role-based access control for different API endpoints or data domains.

- **Testing and Reconciliation:** Beyond payout reconciliation, the system might also do self-reconciliation checks – e.g., verifying that sum of daily revenues equals monthly revenue in total, or that revenue minus expenses equals the change in cash (if linking to actual bank data). These can act as integrity checks and any discrepancies would alert that data might be missing. This is more of a nice-to-have, but worth designing for if completeness is required.

In summary, governance features ensure that the engine's outputs can be trusted and verified. Every number can be backed up by underlying data, changes are tracked, and errors can be corrected systematically. This builds confidence among users (like finance teams) that the unified engine is a reliable source of truth.

Data Schema: Firestore JSON vs Relational Tables

To implement the above, one must choose how to store the data. We'll outline schemas in two paradigms: a NoSQL document model (like Google Firestore, which works nicely with a Next.js/TypeScript stack) and a relational model (like PostgreSQL or BigQuery with SQL tables). Each has pros and cons, which we discuss in the next section. Here are example schemas:

Firestore JSON Schema (NoSQL Document Model)

In Firestore, data is organized in collections of documents (JSON-like). We can have separate collections for transactions, subscriptions, payouts, and summary projections. Example structures:

- **Collection:** `transactions` – Each document represents a financial transaction or event. It could include sales, refunds, fees, etc. A possible document (with ID `txn_ABC123` for example) is:

```
{
  "id": "txn_ABC123",
  "type": "sale",
  "date": "2025-09-01T14:30:00Z",
  "amount": 100.00,
  "currency": "USD",
  "fee": 2.90,
  "refund": 0.00,
  "net_amount": 97.10,
  "product_id": "prod_001",
```

```
  "product_name": "Software License",
  "source": "Stripe",
  "order_id": "order_789",
  "customer_id": "cust_123",
  "tags": {
    "category": "Software Sales",
    "region": "NA"
  },
  "payout_id": "po_55555",  // if this transaction is part of a payout
  "description": "Order #789 via Stripe"
}
```

This structure captures a sale of \$100 with a \$2.90 fee. We store both gross `amount` and maybe a computed `net_amount` for convenience (though it can be derived). Including `payout_id` links it to the payout. The `tags` field contains inherited tags – e.g., the system might have auto-tagged this as "Software Sales" category and region "NA". If a user overrides, that could either change this field or add another field like `"user_category_override": "OtherCategory"` which the logic would prefer over the auto category.

Refunds or fees could be separate docs or could be represented as negative amounts or sub-types. Another approach is to have a single `type` field that could be values like "sale", "refund", "fee". A refund transaction might look like:

```
{
  "id": "txn_DEF456",
  "type": "refund",
  "date": "2025-09-05T10:00:00Z",
  "amount": -50.00,
  "currency": "USD",
  "fee": 0.00,
  "refund": 50.00,
  "net_amount": -50.00,
  "product_id": "prod_001",
  "source": "Stripe",
  "original_sale_id": "txn_XYZ999",
  "tags": { "category": "Software Sales", "region": "NA" },
  "payout_id": "po_55558",
  "description": "Refund of Order #790"
}
```

Here `amount` is negative (or we use a separate field to mark it as refund). We include a reference to the original sale (`original_sale_id`) if applicable. The tag remains the same category as the original sale for consistency in reporting net sales.

- **Collection:** `subscriptions` – Each doc for an active subscription or subscription history:

```
{
  "id": "sub_001122",
  "customer_id": "cust_123",
  "plan_id": "plan_monthly_50",
  "start_date": "2025-01-15",
  "status": "active",
  "monthly_amount": 50.00,
  "currency": "USD",
  "cancel_date": null,
  "pause": null
}
```

We might also maintain a subcollection for each subscription's monthly MRR contribution for quick retrieval, or just compute from this.

- **Collection:** `payouts` – Each document is a payout from a platform:

```
{
  "id": "po_55555",
  "platform": "Stripe",
  "date": "2025-09-03",
  "amount": 1000.00,
  "currency": "USD",
  "status": "matched",
  "matched_amount": 1000.00,
  "difference": 0.00,
  "bank_txn_id": "bank_99999",
  "transactions": ["txn_ABC123", "txn_GHI789", "..."],
  "notes": "Matched to Bank deposit on 2025-09-05"
}
```

This shows a Stripe payout of \$1000 on Sept 3 which has been matched to a bank transaction. We list the transaction IDs included (this could be a lot, but Firestore can handle arrays of IDs; if it's huge, an alternative is to not store them here but derive by querying `transactions` by payout_id – which Firestore can do with an index). For unmatched payouts, `bank_txn_id` might be null and status "unmatched", and maybe a field `expected_transactions_count` to know how many transactions should be there.

- **Collection:** `revenue_summary_day` – Each document for a date (we can use the date as the doc ID, e.g. "2025-09-01"):

```
{
  "date": "2025-09-01",
  "gross_revenue": 15000.00,
```

```
  "net_revenue": 14500.00,
  "orders": 320,
  "units": 340,
  "aov": 46.88,
  "by_product": {
    "prod_001": { "name": "Software License", "gross": 10000.00, "net":
9700.00, "orders": 200, "units": 200 },
    "prod_002": { "name": "Hardware Widget", "gross": 5000.00, "net":
4800.00, "orders": 120, "units": 140 }
  },
  "by_source": {
    "Stripe": { "gross": 12000.00, "net": 11600.00, "orders": 270 },
    "PayPal": { "gross": 3000.00, "net": 2900.00, "orders": 50 }
  }
}
```

This example doc shows the totals for the day and nested breakdowns by product and source. Nesting makes it convenient to fetch one document and get everything needed to, say, draw a stacked chart by product for that day. However, large nested structures can hit Firestore document size limits if there are too many products. If that's a concern, an alternative is a subcollection, e.g., `revenue_summary_day/{date}/products` with docs for each product's total. Simpler still, one might decide not to nest at all and instead query a separate collection `revenue_by_product_day` when needed. But for moderate dimensionality, nesting is fine.

- **Collection:** `expense_summary_month` (could similarly have day). For month, doc ID like "2025-09":

```
{
  "month": "2025-09",
  "gross_expenses": 8000.00,
  "net_expenses": 7500.00,
  "by_category": {
    "Marketing": 2000.00,
    "Salaries": 3000.00,
    "Software": 500.00,
    "Travel": 1500.00,
    "Miscellaneous": 1000.00
  },
  "by_account": {
    "BankChecking": 5000.00,
    "CorporateCard": 3000.00
  },
  "anomalies": [
    { "category": "Travel", "expected": 500.00, "actual": 1500.00,
"z_score": 2.5 }
  ],
  "top_vendors": [
```

```
    { "name": "Google Ads", "total": 1800.00 },
    { "name": "Facebook Ads", "total": 1500.00 },
    { "name": "Delta Airlines", "total": 1200.00 }
  ]
}
```

This monthly expense doc sums to \$8000 gross (i.e. actual payouts) but net is \$7500 after some refunds/credits. It breaks down by categories (Marketing, etc.) and by account (just examples of how multi-dimensional it can be). It also lists an anomaly: Travel was much higher than expected (maybe based on seasonal expectation), which the engine flagged. And it lists top vendors by spend for that month. This kind of data is directly what a dashboard would display (e.g. pie chart of categories, list of top vendors, etc.), meaning the API can just serve this JSON.

- **Collection:** `cohort_summary` – If storing cohorts in Firestore, perhaps one document per cohort start month:

```
{
  "cohort": "2021-05",
  "customer_count": 11,
  "mrr": 1100.00,
  "retention": {
    "month_0": 100.0,
    "month_1": 100.0,
    "month_3": 90.9,
    "month_6": 72.7,
    "month_12": 63.6
  }
}
```

This shows percentages of the original cohort remaining at various milestones. Alternatively, store an array like `"retention_curve": [100, 100, 90.9, ... 63.6]` where index corresponds to month 0,1,2... etc. The advantage of a structured object is clarity. The disadvantage is if we want to query, say, "what's the average 6-month retention across cohorts" we'd have to either compute in code or redundantly store it in a separate structure.

The above JSON schemas are illustrative. Firestore is schema-less in that we don't have to define this ahead of time, but we conceptually maintain this structure. We also likely want composite indexes (like on `transactions` by `date` and maybe `tags.category`) to query quickly. Firestore would be very good at retrieving a single doc by key (e.g. get the summary for a specific day or specific payout by ID). It can also perform range queries (like all transactions in a date range) but those need careful indexing and can become expensive if the dataset is huge (BigQuery or SQL might handle large scans better for analytics).

Relational Schema (SQL tables)

In a relational model, we define tables with fixed schemas and use foreign keys for relationships. An SQL schema for our engine could look like:

- **Table** `transactions`**:** Each row is a financial transaction/event. Key columns:
- `id` (primary key, e.g. varchar or UUID)
- `date` (timestamp or date)
- `type` (enum or text: 'sale', 'refund', 'fee', 'chargeback', 'adjustment', etc.)
- `amount` (numeric) – positive for sales, negative for refunds/credits perhaps, or always positive and use type to indicate sign
- `currency` (varchar(3))
- `fee_amount` (numeric, default 0) – any fee associated with this transaction (for a sale, the processing fee; for a fee type transaction, it might equal amount; for expense, could be blank)
- `net_amount` (numeric) – convenience or computed as amount - fee_amount - refund_amount (if those are separate)
- `product_id` (varchar) – foreign key to a `products` table if product metadata is stored, or could store product name directly if denormalizing
- `order_id` (varchar) – if relevant (for revenue, link to an order/invoice)
- `customer_id` (varchar) – if needed for deeper analysis
- `subscription_id` (varchar) – if this transaction is part of a subscription (like a monthly charge), link it
- `payout_id` (varchar) – foreign key to `payouts` table if applicable (for revenue transactions linked to a payout)
- `expense_vendor` (varchar) – for expense records, could have vendor name or ID
- `account` (varchar) – for expense records, the account/credit card used
- `category_tag` (varchar) – a primary category tag. In SQL, multi-valued tags are tricky; one approach is to have a separate table for tags (many-to-many). But for simplicity, if we assume one main category per transaction, store it here. For multiple tags, a separate `transaction_tags` junction table (transaction_id, tag) would be needed.
- other possible columns: `description` (text), `original_txn_id` (to link a refund to the original sale), etc.

- Indices: likely index on `date`, on `payout_id`, on `category_tag`, etc., to speed up typical queries.

- **Table** `payouts`**:** Columns:

- `payout_id` (primary key)
- `platform` (Stripe, MercadoPago, etc.)
- `date` (date the payout was created or paid)
- `amount` (numeric)
- `currency`
- `status` (enum: 'unmatched','matched','partial')
- `bank_transaction_id` (if matched, reference to some bank transactions table or an identifier of bank statement line)
- `difference` (numeric, any leftover difference)

- Possibly `expected_count` or `expected_amount` if we want to log what we expected vs got.
- We wouldn't list transactions here because in SQL we can just join transaction table to get them (SELECT * from transactions WHERE payout_id = X).

- **Table `subscriptions`:** (for active subscriptions, if needed aside from using transactions)

- `subscription_id` (PK)
- `customer_id`
- `plan_id` or details about plan (maybe join to a plans table)
- `start_date`, `end_date` (or cancel_date)
- `status`
- Perhaps `monthly_amount` or keep plan info separately.
- This table helps to compute MRR by summing active subs. Alternatively, one could simply use the transactions table to derive MRR (e.g. for each month, sum all subscription charges or look at who was charged). But having the subscription terms can help to predict future MRR or see churn without waiting for charges.

- **Table `products`:** If needed, containing product metadata like name, category (for inheritance of tag maybe). If product has a category, when joining transactions to products you get the category tag.

- **Table `tags`:** If implementing a flexible tag system in SQL, a `tags` table and a `transaction_tags` linking table could be used. However, since many tags (category, region, etc.) can also be modeled as columns or foreign keys to dimension tables, one might simplify by having dedicated columns for known dimensions (e.g. category_tag as above, region, department, etc.) and use a general tag table only for less structured labels.

- **Summary Tables:** We create tables for the read models:

- `revenue_summary_day`: Columns: `date` (PK or part of PK if also splitting by product), `gross_revenue`, `net_revenue`, `orders`, `units`, `aov`. If we want to precompute breakdown by product in SQL, we have two options: either include `product_id` in this table's primary key (so each row is per date per product), or have a separate table `revenue_by_product_day`. The decision depends on usage. If queries often filter by product, a table indexed by product and date is useful. Let's assume we want both overall and per product. We can have:
  - `revenue_summary_day` table with one row per date (overall totals).
  - `revenue_summary_product_day` table with date, product_id, gross, net, orders, etc. (So to get breakdown for a date range, one would query this table with a WHERE date between X and Y, possibly pivot in the app or just group by product).

  Similarly, `revenue_summary_month` (and `revenue_summary_product_month` if needed).

- `expense_summary_month`: If expenses are usually viewed monthly by category, we make a table: columns: `month`, maybe `category` as part of PK if per category. Or just store one row per month

with a JSON or array of category breakdowns (but SQL doesn't handle that well for querying). Better to have:

    - `expense_summary_month` (overall totals per month)
    - `expense_summary_category_month` (per category per month totals). And possibly `expense_summary_account_month` for account breakdown.

- `payout_reconciliation` view or table: This could actually be a **view** that joins payouts and transactions to calculate match statuses live, but to incorporate tolerances and partial matches it might be easier as a table updated by the reconciliation process. It might mirror the `payouts` table with a bit more info, or simply be the `payouts` table itself with status fields filled (so maybe we don't need a separate table if we add status, difference to `payouts` table as above). Each payout row effectively is our reconciliation record. Additionally, if we want to track unmatched bank transactions (deposits that have no corresponding payout), those could either be in another table or we treat the bank statement as another source and compare. That might be beyond scope; we focus on payouts from systems.

- `cohort_summary`: A table with `cohort_month`, `month_number`, `customers_remaining`, `MRR_remaining` or `retention_rate`. Each cohort-month combination can be a row. e.g. cohort=2021-05, month_number=0, customers=11; cohort=2021-05, month_number=12, customers=7 (63.6% retention). This is a normalized way that's easy to query for trends (like average retention at month 3 across all cohorts starting in 2021, etc.). To fetch a single cohort's curve, you SELECT all rows for that cohort ordered by month_number. Alternatively, store each cohort as one row with many columns (month0_rate, month1_rate,...). That's a wide schema which is less flexible but fast to fetch all at once. Given SQL's column limit might handle 24-36 months easily, that's also an option. One could also use an JSON column to store an array of retention values for the cohort. Each approach has tradeoffs in queryability vs simplicity.

The relational model tends to require more upfront design but makes certain queries straightforward. For example, summing revenue by product by month is a simple GROUP BY on transactions or on the summary table. In Firestore, you'd either store it ready-made or run multiple queries client-side (which is possible but less efficient for big data).

We will now briefly discuss which storage to choose.

Architectural Tradeoffs: Firestore vs. BigQuery vs. Postgres

Choosing the right storage and processing engines is critical and depends on scale, expertise, and use cases:

Firestore (NoSQL, document store): This is a good fit if the application is already on Firebase/Firestore (e.g. using it as a backend for Next.js). Firestore excels at real-time updates and simplicity of getting a document by key. It's schema-flexible, which suited our JSON designs above. It can also scale to a large number of documents, and you pay for what you use in terms of reads/writes. Firestore would allow us to easily update a specific summary document when a new transaction comes in (within a transaction to avoid race conditions). It also integrates nicely with serverless functions. However, Firestore has limitations: querying is less powerful than SQL (you can't do arbitrary aggregations on the fly; you typically store the

aggregation results as we planned). Complex queries with multiple filters require composite indexes and are limited (for instance, it's not trivial to ask Firestore "give me total sales by month for each product" without having pre-stored those totals or doing client-side aggregation of multiple queries). Firestore is great for **operational workload** – quick reads/writes of specific items – but not as strong for heavy analytical queries scanning lots of data. Another consideration: multi-document transactions in Firestore are possible but only up to 500 documents or so at a time; in our design we mostly update one summary at a time so that's fine.

PostgreSQL (or another relational database): A traditional relational database is very good for ensuring consistency (ACID transactions), complex querying (SQL joins, group by, etc.), and enforcing schema (which can catch data errors early). Postgres can handle moderate data volumes on a single instance, and can be tuned or scaled vertically if needed. It also supports extensions (like time-series extensions or JSON fields if semi-structured needed). Using Postgres, one could compute summaries either via SQL queries or using something like Materialized Views or triggers to maintain summary tables. The tradeoff is the complexity of managing the database (backups, scaling connections) and possibly performance at large scale (millions of transactions) unless using partitions or an analytics DB extension. If the data size is not huge (say <10 million transactions), Postgres might handle it fine and gives the advantage that you can do ad-hoc queries if new reporting needs come up (whereas in Firestore you'd have to create new structures or export the data to analyze). Postgres also allows using tools like stored procedures or CTEs to encapsulate some of the transformation logic – but with our approach, we might let the application or dbt handle that instead.

BigQuery (or Snowflake, Redshift, etc. data warehouses): BigQuery is a serverless data warehouse that excels at scanning large data sets quickly for analytic queries. If our engine needs to handle very large volumes (hundreds of millions of rows, or lots of complex aggregation queries ad-hoc), BigQuery is a strong choice. For instance, if we want to allow the user to dynamically ask "show me sales by state by week for the last 5 years", BigQuery can compute that on the fly by scanning fact tables, whereas Firestore cannot (unless that specific aggregation was precomputed). BigQuery also integrates with BI tools easily. The downside is BigQuery queries have a cost (per TB scanned) and a slight latency (usually a couple seconds per query), which might be fine for analytics dashboards but could be too slow for a highly interactive UI where you expect sub-second responses. Another downside is BigQuery is eventually consistent on streaming inserts and doesn't support transactions across multiple tables – which might complicate an event-driven update to multiple summary tables. Typically, with BigQuery, one would recompute summaries in batch (maybe using scheduled queries or via dbt). So BigQuery is ideal if the use case is more "big data analytics and periodic reporting" rather than real-time interactive use by many concurrent users. In our scenario, if the business has high volume and needs complex slicing/dicing beyond what we planned, BigQuery could be an appropriate backbone. In fact, a hybrid approach could be: use Firestore for storing the latest summaries for quick UI access, but periodically dump data to BigQuery for deeper analysis or auditing purposes (Firestore has an integration to stream data to BigQuery for analytics).

Other considerations: There are other data stores like Amazon DynamoDB (similar to Firestore in many ways), or even using an OLAP cube (like Pinot or ClickHouse) if extremely fast aggregate queries are needed on large data. But those add complexity.

Comparison Summary: Firestore is simple and real-time but less analytical; BigQuery is powerful for analytics but more batch-oriented; Postgres is a middle ground of consistency and query power but might require more upkeep.

Given our target implementation mentions Next.js/TypeScript, which often pairs well with cloud-native services, one strategy could be: start with Firestore for quick development and real-time updates, and if needed, complement with BigQuery for heavy analysis and use dbt to manage the transformations in BigQuery. Firestore could hold the current state (for the app's quick needs) while BigQuery can be the sink for all raw events and do number crunching at scale if needed (for internal analysts or for sanity checks). Indeed, some companies use both: for example, they maintain a "source of truth" in a warehouse but use a fast NoSQL or cache for serving the application. This is akin to the separate read store in CQRS [28] [29], where perhaps Postgres/Firestore is the operational read DB and BigQuery is the analytical deep-dive store.

If we compare to **open-source/vendor systems**: a combination of Airbyte (for extraction) + dbt (for transformation) + BigQuery (warehouse) + our custom Next.js app (for presentation) is a very modern stack [18] [19]. In contrast, using Firestore + custom code is more custom but might reduce reliance on big third-party components.

We should also mention cost: Firestore cost is based on operations (reads/writes), BigQuery cost is based on data scanned and storage, Postgres cost is typically a fixed cost (the server or managed instance). If expecting heavy usage, BigQuery might incur high query costs, whereas Firestore might be cheaper for predictable access patterns, or vice versa depending on data size.

In summary, the decision may come down to scale and need for flexibility. For a **fast iteration and integrated app** feel, Firestore is attractive. For **robust analytics and future-proofing** with SQL, BigQuery or Postgres is better. Often, using both isn't a bad idea: do initial processing in SQL (where complex joins are easier), then export final results to Firestore for the app to use. This hybrid approach gives best of both (and tools like Cloud Functions or Cloud Run can link them).

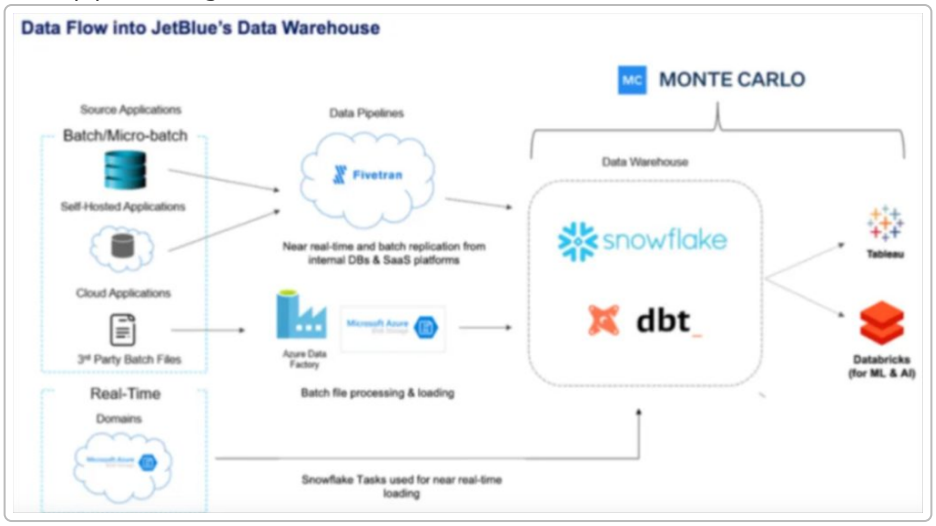
Integration with Existing Tools and Systems

It's useful to compare our design with existing open-source or vendor solutions and see how we can leverage them:

- **dbt (Data Build Tool):** dbt is an open-source framework for managing SQL transformations (especially on warehouses like BigQuery/Snowflake/Postgres). Instead of writing raw SQL in a vacuum, you create dbt models (which are basically select queries) and dbt materializes them as tables or views, handling dependencies. In our case, we could implement all our summary computations in dbt SQL models. For example, a model for `revenue_summary_day` that selects from raw transactions, or a model for `payout_reconciliation` that joins transactions to payouts. dbt ensures idempotence by default (it will typically `CREATE OR REPLACE` tables) [30] and it can be scheduled to run incrementally for new data. The benefit is maintainability – complex SQL logic can be broken into parts (like one model for computing net transaction values, another that aggregates by day, etc.), tested, and documented. If our team is comfortable with SQL, dbt would be a great addition to implement the transformation layer. It also integrates with version control and CI, which is good for governance (you can peer review changes to how metrics are computed). Some of the heavy lifting described (e.g. currency conversion, MRR calc) could be expressed in SQL and let the warehouse handle it. However, not everything is trivial in SQL (cohort analysis in pure SQL is possible but can be complex; though one could also do it with window functions or recursive CTEs). If not using dbt, we might end up writing similar SQL in our application or performing calculations in

TypeScript. Given that financial calculations benefit from set-based operations, using dbt/SQL is likely more efficient and less error-prone than custom code for large datasets.

- **Airbyte (or Fivetran):** These are data integration tools that provide connectors to various sources. For instance, Airbyte has a Stripe connector that can fetch all Stripe charges, refunds, payouts, etc., and load them into a destination like Postgres or BigQuery. Instead of writing and maintaining code to call Stripe's API, handle pagination, webhooks, retries, etc., we could configure Airbyte to continuously sync Stripe data into our database. Similarly, an Airbyte connector could fetch bank transactions (maybe via Plaid or just CSV import). Using such tools offloads a lot of ETL work. The unified engine we're building would then start from the data that Airbyte has already loaded. It aligns with our pipeline diagram where Fivetran was used



– Airbyte is an open-source alternative. Airbyte can also normalize data (though often you still need to define transformations after loading). The tradeoff is relying on another service and possibly less real-time than hitting a webhook. But Airbyte does support incremental syncs and even webhook-based intake for some sources. If the goal is not to reinvent the wheel, one could let Airbyte populate the `transactions` and `payouts` tables from Stripe/Mercado Pago, and then our engine's job is mainly to aggregate and present (something dbt and our Next.js UI can handle).

- **Stripe API/SDK:** If not using a generalized tool like Airbyte, one can use Stripe's official SDK (available for Node/TypeScript) to pull data. Stripe's API allows listing all charges, all balance transactions, all payouts, etc. One has to be careful to reconcile them properly (Stripe has objects for Charges and a separate BalanceTransaction for each movement of funds which includes payouts, fees, etc.). Stripe also provides webhooks (e.g., an event when a payout is paid, when a charge is created, etc.). A robust design might combine webhooks (to get near real-time push of events) with periodic full syncs (to correct any missed events or backfill historical data). The advantage of using the SDK directly is control and possibly cost savings (no third-party) but the disadvantage is the development and maintenance effort. Also, consider other platforms: Mercado Pago also has APIs for transactions and settlements – those would require separate integration code.

- **Stripe Sigma / Reports:** Stripe Sigma is a built-in SQL-based reporting tool in Stripe's dashboard, and Stripe also can provide payout reconciliation reports. However, using Sigma or Stripe's own reports is more of a manual or separate process – our engine aims to unify across platforms. But it's

worth noting: if one only cared about Stripe revenue, Sigma might do many of these metrics out-of-the-box (they have MRR, cohort reports in Stripe if using Stripe Billing). The value of our approach is it's platform agnostic and customizable. Similarly, for expenses, accounting software (like QuickBooks or Xero) can produce expense reports, but if the data is spread or needs custom analysis (like our anomaly detection), a custom engine adds value.

- **Open-source Finance dashboards:** There are some open-source projects and libraries. For example, **Ghostfolio** (personal finance tracker) or other budget tools might have components like recurring expense detection, but those are usually simpler and targeted at personal finance. Our scope is more business-oriented and comprehensive.

- **Comparison to ERP systems:** A unified engine has some overlap with what an ERP/accounting system does (record all transactions, produce financial statements). We are not fully replacing an accounting system here, but indeed some features (tagging, summarizing) are akin to an accounting General Ledger and reporting module. For a small business, this engine might suffice for managerial reporting, but for compliance they'd still use an accounting system. One might integrate by exporting our results to the accounting system or vice versa. The design here is more flexible with tags and custom metrics, whereas an accounting system might have a rigid chart of accounts. We even referenced a *Unified Chart of Accounts (UCOA)* concept in search results, which is about standard categorization for comparison [31] – our tag model could implement a custom chart of accounts.

- **Vendor solutions for subscription analytics:** e.g. ProfitWell (now part of Paddle), ChartMogul, Baremetrics – these are specialized SaaS metrics dashboards which automatically calculate MRR, churn, cohorts when you feed them subscription data. They are essentially doing what our subscription part does. We compare in the sense that those are ready-made but only for subscriptions (and require using certain payment gateways). Our engine is more general (combines expense and other revenue). If one had the budget and less need for customization, they might use ProfitWell for MRR and an accounting BI for expenses, but the unified approach gives one coherent view and the ability to tie things together (like net revenue minus expenses to see profit, etc., in one system).

- **Machine Learning for Anomalies:** If we were to extend, there are open libraries for anomaly detection (like Twitter's Anomalydetection or using Python's statsmodels for STL, etc.). One could incorporate those via a small job that runs each month's expense data through an algorithm to flag anomalies. We already incorporate the concept using z-scores and STL conceptually, but hooking up a library could refine it. It's beyond our scope to fully implement ML here, but good to design for the possibility (perhaps mark certain series of data and feed to an anomaly detection microservice). Similarly for forecasting (predicting next month's revenue or expenses), which often goes hand-in-hand with such engines for budgeting purposes.

In conclusion, our design does not reinvent proven patterns: it aligns with the modern data stack approach (extract with connectors, load to warehouse, transform with SQL, present via a specialized UI) [18] [19]. The decision to build a custom Next.js app for UI suggests a need for interactive and company-specific views that generic BI tools may not provide easily – but we could still leverage the mentioned tools under the hood for heavy lifting. By understanding these comparisons, we ensure our architecture is compatible with or can integrate these tools rather than conflict with them (for example, we could use dbt to manage our

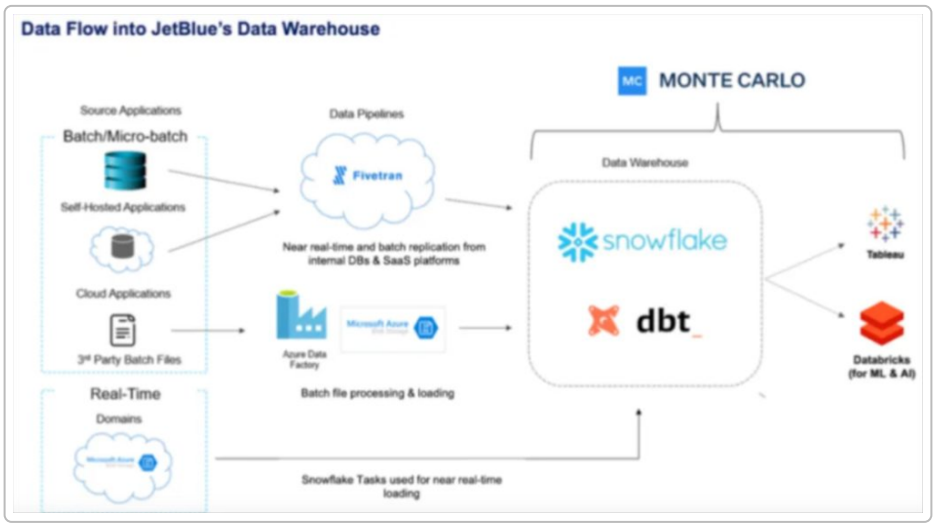
SQL, and nothing in our design prevents that – it in fact encourages idempotent, rebuildable transforms which is dbt's philosophy [23]).

Conclusion and Summary Diagrams

We have outlined a comprehensive design for a unified revenue and expense analytics engine, covering everything from data ingestion to metrics computation and API outputs. By leveraging CQRS principles and thoughtful data modeling, the engine can provide a real-time, trustworthy view of a business's finances across revenue, subscriptions, and spending.

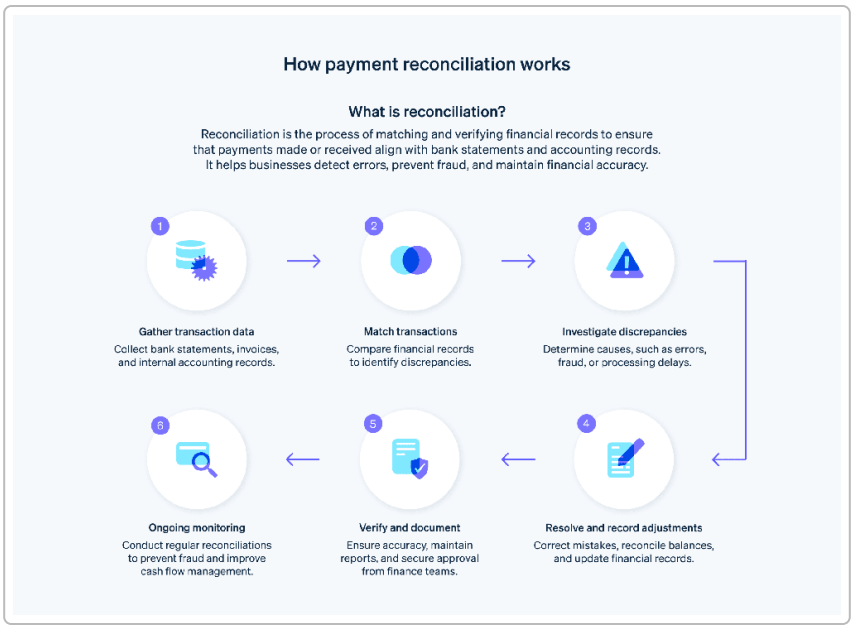
To recap key points in a few diagrams:

- *Data Flow / Pipeline:* Sources (Stripe, Mercado Pago, Bank, etc.) -> Ingestion (APIs or tools) -> Raw Data Store -> Transformation/Aggregation (SQL or code) -> Summary Data Store -> Next.js/React UI & APIs



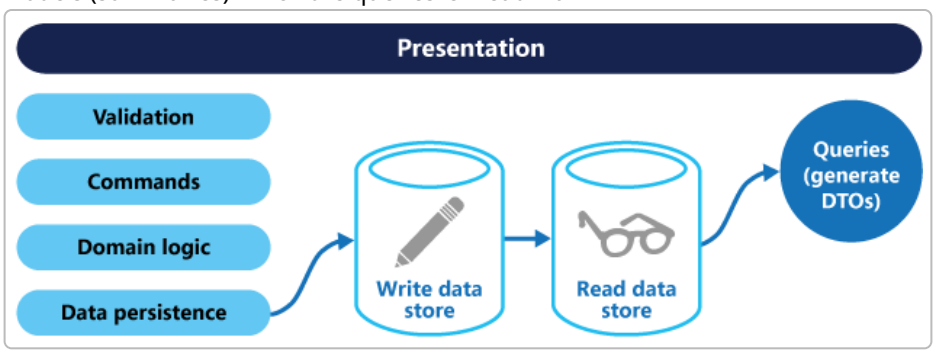
. This pipeline ensures all raw data is collected and then refined into the metrics we need.

- *Reconciliation Process:* Visualized as a cycle of gathering records, matching transactions to payouts, identifying differences, resolving/adjusting, and verifying



. This ensures reported revenue actually equals money in the bank, building trust in the numbers.

- *CQRS and Projections:* Commands (new transactions, updates) go to the write model; events update the read models (summaries) which the queries/UI read from



. This separation means we can optimize each side: writes focus on capturing data, reads focus on fast retrieval of aggregated results.

Finally, we provided pseudocode and schema examples to guide implementation. As a last piece, here is a simplified pseudocode summary tying many components together (in a MapReduce style) for computing daily revenue and detecting an expense anomaly:

```
// MapReduce-style pseudocode for revenue and expense processing

// Revenue Daily Aggregation Map:
for each transaction in transactions:
    if transaction.type in ("sale","refund","fee"):
        dateKey = format(transaction.date, "YYYY-MM-DD")
        // Normalize to base currency
```

```
        amount_base = convert_to_base(transaction.amount, transaction.currency,
transaction.date)
        fee_base = convert_to_base(transaction.fee, transaction.currency,
transaction.date)
        net_base = amount_base - fee_base - (transaction.refund_base or 0)
        emit (key: ("revenue", dateKey, transaction.product_id,
transaction.source),
              value: { gross: amount_base, net: net_base, orders:
(transaction.type=="sale"?1:0), units: transaction.quantity or 0 })

// Revenue Reduce:
for each key (date, product, source):
    aggregate gross_sum, net_sum, orders_sum, units_sum
    gross_sum = sum(value.gross for all values)
    net_sum = sum(value.net for all values)
    orders_sum = sum(value.orders)
    units_sum = sum(value.units)
    AOV = gross_sum / orders_sum (if orders_sum > 0)
    store_in_revenue_summary_day(date)[product][source] = {...}  // nested or
separate tables as discussed
    also update overall product and date totals (e.g.,
revenue_summary_day(date).gross += gross_sum, etc.)

// Expense Anomaly Detection (using simple stats):
for each expense_category:
    timeseries = get_monthly_totals(expenses, category=expense_category,
last_N_months=12)
    avg = mean(timeseries)
    stddev = std_dev(timeseries)
    current = timeseries[last]  // most recent month
    if (current - avg) > 3 * stddev:
        emit_alert("Outlier: category " + expense_category + " for this month,
value " + current + " is unusually high")
```

The above pseudocode is illustrative: it shows how each transaction is converted and aggregated, and how an anomaly check might occur for expenses. In a real implementation, one would use SQL or a programming language instead of this pseudo loop.

In conclusion, the unified engine combines elements of data engineering, financial accounting, and web development to deliver a tailored view of a company's financial health. By carefully designing data models (with tags and inheritance), handling multi-currency properly [1], and using proven patterns for data processing (CQRS, idempotent transformations [20] [21]), the system will be maintainable and extensible. The comparisons to other tools suggest our approach is aligned with modern best practices and can integrate or evolve with them. Implemented in Next.js/TypeScript with a UI in React/Tailwind, the final product would be a responsive web dashboard where users can slice revenue by product or channel, monitor KPIs like MRR and churn, see where money is going, and ensure everything is reconciled – all with confidence in the accuracy and lineage of the data driving those insights.

1 2 8 9 The challenges of multi-currency reconciliation | Aurum Solutions
https://aurum.solutions/resources/the-challenges-of-multi-currency-reconciliation

3 A Payment Reconciliation Guide That Prevents Month-End Chaos
https://www.rapyd.net/blog/payment-reconciliation/

4 6 Payment reconciliation: What it is and how it’s done | Stripe
https://stripe.com/resources/more/payment-reconciliation-101

5 7 10 11 Mastering Stripe Payout Reconciliation: Ensuring Accurate Financial Management -
https://www.cointab.net/us/mastering-stripe-payout-reconciliation-ensuring-accurate-financial-management/

12 13 14 Monthly recurring revenue (MRR) explained | Stripe
https://stripe.com/resources/more/what-is-monthly-recurring-revenue

15 16 Understanding your MRR Cohorts Report - Help Center - Paddle
https://www.paddle.com/help/profitwell-metrics/analyse/subscription-metrics/understanding-your-mrr-cohorts-report

17 Inside Tink: how we work with outlier detection | Tink blog
https://tink.com/blog/product/outlier-detection-categorisation/

18 19 Data Pipeline Architecture Explained: 6 Diagrams And Best Practices
https://www.montecarlodata.com/blog-data-pipeline-architecture-explained/

20 21 22 23 26 30 Understanding idempotent data transformations - Archive - dbt Community Forum
https://discourse.getdbt.com/t/understanding-idempotent-data-transformations/518

24 25 27 28 29 CQRS Pattern - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs

31 Understanding Unified Chart of Accounts (UCOA) - SaasAnt
https://www.saasant.com/blog/unified-chart-of-accounts/