ChatGPT

Observability & Data Quality Implementation Guide for the Financial Intelligence App

System Architecture for Observability & Data Quality

Overview & Data Flow: The app's data pipeline is instrumented with observability and DQ checks at every stage, ensuring issues are caught early to maintain trust in analytics (notably, **70% of business leaders lack trust in dashboards due to frequent data quality incidents** [1]). The single-user (yet SaaS-ready) architecture treats each user as a tenant with isolated data and DQ tracking. Data flows through **ingestion**, **normalization**, and **intelligence** layers, with quality checkpoints at each stage: - **Ingestion Checkpoint:** Raw inputs (bank transactions, credit card statements, user CSV uploads, etc.) are validated on arrival. - **Normalization Checkpoint:** After data is cleaned and standardized (e.g., mapping categories, formatting dates), integrity checks run on the normalized tables. - **Projection Checkpoint:** The Intelligence Layer generates read-optimized projections (aggregated stats, forecasts, budgets, etc.), which are verified against source data for consistency. - **AI/Model Output Checkpoint:** Any AI-generated outputs (e.g. forecasts, categorizations) undergo quality checks comparing predictions to actuals or expected patterns.

DQ & Observability Components: The system comprises dedicated microservices for data quality and observability, each focusing on part of the monitoring workflow:

- **DQ Runner:** Orchestrates execution of data quality rules at each checkpoint. It triggers on events (e.g., new data ingestion or pipeline job completion) and on schedules. The DQ Runner evaluates validation queries and rules (see DQ Rule Catalog) and writes results to the `dq_results` store. It handles severity logic (e.g., abort pipeline on *ERROR*, log and continue on *WARN*).

- **Lineage Writer:** As data flows through transformations, the Lineage Writer records lineage links (source-to-target mappings) in the `lineage_edges` table. Every derived record or aggregate knows its parent sources, enabling full traceability. This lineage metadata is crucial – **when a data issue is detected, lineage serves as a "map" to pinpoint where it originated** [2]. For example, if a Dashboard total is off, lineage edges can trace it back to a specific faulty transaction in the raw data. The **provenance** table complements this by capturing source context (e.g. file name, source system, ingest timestamp) for each raw record, so any anomaly can be tracked to its original input.

- **Reconciliation Engine:** A specialized service for cross-system checks. It compares records across modules for consistency – e.g., ensuring the sum of **Expenses** matches the outflow in **Bank** accounts within a tolerance, or that credit card payments in the **Credit Card** module align with withdrawals in **Bank**. The recon engine writes pairing results to `reconciliation_links`. It flags discrepancies beyond a defined tolerance (e.g. >1.5% mismatch in account totals) for investigation. These automated reconciliations run after each data load and at period close (month-end) to catch integration issues between financial sources.

- **Metrics Gateway:** All metrics and signals from the pipeline feed into the Metrics Gateway. It aggregates data quality metrics (row counts, schema changes, distribution stats, model errors, etc.) and pushes them to both the internal database (tables like `forecast_metrics`, `categorization_metrics`) and an alerting system. This enables both historical analysis and real-time monitoring. For instance, the Metrics Gateway records that the latest forecast's SMAPE error was 12%, and if this exceeds the threshold, the DQ Runner will mark a rule violation. It also collects operational telemetry (latency, volume) to watch for anomalies like a sudden drop in transactions ingested (a possible missing data issue).

- **Incident Bot:** The incident management automation. It monitors the `dq_results` table and metrics for critical issues. When a high-severity rule fails (or a WARN remains unresolved beyond a grace period), the Incident Bot creates an incident ticket or sends an alert. For example, if an **ERROR** occurs in the **Forecast** module (forecast error too high) or a **Budget** integrity check fails, the bot might notify the user and log an incident in a tracker. It can also escalate issues: e.g., send an email or chat notification for immediate attention. The Incident Bot helps ensure no quality issue goes unnoticed by the user.

Data Quality Storage & Schema: All DQ outputs are stored in a dedicated schema (or database) separate from business data. Tables like `dq_results`, `lineage_edges`, etc. are **scoped by userId to support multi-tenant SaaS** – each user's DQ data is isolated. (In practice, this could mean each user gets a private schema or dataset for quality results; for example, DQOps creates a **private dataset per user to store all check results and incidents** [3] [4].) This design ensures observability at scale without cross-user data leakage.

Data Flow with Checkpoints: During operation, as data moves through the 12 modules: 1. **Ingestion Stage:** e.g. a bank statement CSV is uploaded. The ingestion service writes raw records to a staging table. The DQ Runner immediately triggers **ingestion rules** (format checks, field completeness, duplicates, etc.). Results (pass/fail) go to `dq_results`. If any *ERROR* (e.g., file unparseable), the Incident Bot can halt processing and alert, whereas *WARN*s (e.g., a few duplicate transactions) are logged for later review. 2. **Normalization Stage:** The raw data is transformed (dates parsed, descriptions cleaned, categories assigned via the Category Mapper module, etc.) into canonical tables. The DQ Runner then executes **normalization rules** (e.g., no null critical fields, valid category values, referential integrity) on these tables. 3. **Intelligence/Projection Stage:** The Intelligence Layer reads normalized data to produce projection tables – e.g., monthly aggregates for the **Dashboard/Charts**, forecast outputs for **Forecast** module, computed budget utilization for **Budget**, category heatmap data for **Heat Map** module, etc. After generating these, the DQ Runner runs **projection checks**. For example, it verifies that the **Revenue** projection for August equals the sum of all August transactions in the detail table (ensuring no loss in aggregation), or that a **Heat Map** has data coverage for all expected time buckets. 4. **AI Model Outputs:** The AI Layer (ML models for forecasting, anomaly detection, categorization suggestions) adds another checkpoint. The system logs model metrics (accuracy, drift) to `forecast_metrics` and `categorization_metrics`. The DQ Runner evaluates **model-specific rules** like forecast accuracy thresholds and data drift. For instance, if forecast SMAPE exceeds 10%, it records a WARN in `dq_results` and the Incident Bot might prompt retraining. *Forward-looking:* This design anticipates **model drift monitoring** – e.g., computing Population Stability Index (PSI) on input data distributions over time. If a drift rule triggers (PSI > 0.2), it's logged and can prompt model updates (indeed, a PSI $\geq$0.2 is a common trigger indicating significant population change requiring model re-evaluation [5] [6]).

Lineage & Traceability: At each transformation, the Lineage Writer appends entries in `lineage_edges`. This creates a graph of how data travels: e.g., a raw **Bank** transaction (ID `tx123`) → normalized transaction (ID `n456`) → aggregated monthly total (ID `m789`). If the monthly total fails a DQ check, one can traverse lineage edges to find which raw `tx123` (or others) contributed, and examine their provenance (say `tx123` came from `January.csv` line 42). **Every DQ rule violation can thus be traced back to its source inputs via lineage**, fulfilling a core observability requirement that when something is wrong, we can diagnose the root cause [2]. This is further aided by `provenance` metadata (e.g., `tx123` originated from *ChaseBank API* on 2025-09-01). Such traceability not only helps debugging but also **enables automated fixes** – knowing the source means the system (or user) can correct data at the origin.

Integration with Change-Set Ledger: The app maintains a Change-Set ledger of data modifications. The Observability system ties into this for continuous improvement. When the DQ Runner logs a rule violation as a *WARN* that remains unresolved (not fixed by the user or automatically), the Incident Bot will spawn a **change proposal** in the ledger. For example, if a **categorization** rule finds transactions labeled "Miscellaneous" spiked (category drift), a proposal might be created suggesting to split or reassign categories. If an **account reconciliation** is off by $50 (within tolerance so not an ERROR but a WARN), a proposal could suggest an adjusting entry or investigation task. These change-set entries serve as actionable to-do's derived from data quality insights. They ensure that data issues are not only observed but also tracked to resolution in the same workflow that governs manual changes. This tight integration means the system not only monitors quality but also drives a remediation workflow.

AI Layer Feedback Loop: The AI Layer consumes DQ context to enhance the user experience. It uses the `dq_results` and metrics as feedback: - For any *auto-fixable* rule (marked Y in our catalog), the AI can attempt a direct fix. For instance, if a **normalization** rule says a field is missing and Auto-fix=Y, the AI might fill it (perhaps using an inference or default). - For issues requiring user input, the AI assistant can **suggest fixes**. E.g., "Transaction on 2025-09-10 has no category – I suspect it's a *Utilities* expense based on description. Shall I categorize it as Utilities?" or "Forecast error is high; consider retraining the model with the latest 6 months data." - The AI also uses DQ lineage to explain anomalies: e.g., "Your Dashboard total for Q3 is lower than expected because 5 transactions were flagged as duplicates and excluded [7]. You might want to review those duplicates."

This AI-driven remediation is informed by DQ context – effectively closing the loop from detection to resolution. Moreover, any AI-proposed changes go through the Change-Set ledger (so they are auditable and can be accepted or rejected by the user).

Future-Proofing (Versioning & Deltas): The observability design anticipates growth and more complex needs: - **Time-based Versioning (`asOf` queries):** All data changes and DQ results are timestamped. The combination of lineage and the change ledger means we can reconstruct historical states. For example, each projection can be tagged with an `asOf` date/version. Users could query "Budget status as of 2025-09-01" and the system, using the ledger of changes and data snapshots, can provide a consistent view. Internally, important tables (normalized data, projections) may implement **slowly-changing dimensions or append-only logs** such that nothing is hard-deleted – instead, records get validity intervals (`valid_from`, `valid_to`). This allows the observability system to evaluate data quality retroactively on prior versions if needed, and to retain **evidence** for any historical incident. - **Checksum-Based Delta Rebuilds:** To optimize performance at scale, the pipeline computes checksums for data segments and uses these to decide what to recompute. Each ingestion batch (or file) gets a checksum in the `provenance` table; each normalized dataset and projection may get a rolling checksum. When new data arrives or changes, the pipeline

identifies which downstream projections are impacted via lineage, and only those with changed inputs are rebuilt. For example, if the **Expenses** for January haven't changed (same checksum), we skip recalculating the January budget utilization projection. Checksums ensure that even as data grows, recomputation (and re-running DQ checks) can be localized to the changed portions, enabling efficient observability in a SaaS environment.

In summary, the architecture interweaves observability into every step: from robust data contracts on input, to automated checks on processing, to lineage-backed traceability on outputs. Monitoring (via metrics and rules) works in tandem with lineage (for root cause analysis), forming the backbone of trustworthy data operations [8]. Any data quality issues are immediately detected, traced, and funneled into automated or user-assisted remediation, ensuring the financial intelligence app remains reliable and accurate for the end-user.

Data Quality Rule Catalog

Below is a catalog of DQ rules organized by stage and module, with details on each rule's intent and configuration. Each rule has a unique **ID**, a short **Description** of the check, its **Severity** level (*WARN* = does not stop processing, *ERROR* = critical issue), whether it has an **Auto-Fix** policy (Y/N), the owning **Module** (or team) responsible, a sample **Validation Query/Logic**, and an **Example** of what would trigger the rule. These rules cover ingestion and processing checks, financial reconciliation logic, and business-specific validations across the app's modules.

Ingestion Data Quality Rules

- **INGEST-001: Required Fields Present** – **Description:** Validates that all mandatory fields are present in incoming data (no null or empty values in primary fields such as transaction date, amount, accountId). **Severity:** ERROR (data is rejected if critical fields missing). **Auto-fix:** N (cannot auto-fill truly missing core data). **Owner:** Ingestion Service. **Validation:** e.g. for a CSV import, run a schema check or `SELECT COUNT(*) FROM staging WHERE date IS NULL OR amount IS NULL`. If count > 0, fail. **Example:** A bank transactions file is missing dates on some rows – those records are logged and cause an ERROR to prevent ingesting incomplete data.

- **INGEST-002: Data Type & Format Check** – **Description:** Ensures ingested data conforms to expected types/formats (dates in ISO format, amounts are numeric, IDs match UUID regex, etc.). **Severity:** ERROR for unparseable records, WARN for minor formatting issues that can be corrected. **Auto-fix:** Y (for minor issues, e.g., trim whitespace or parse slight format variations). **Owner:** Ingestion Service. **Validation:** e.g., attempt to cast fields to expected data types; in SQL: `SELECT * FROM staging WHERE TRY_CAST(amount AS DECIMAL) IS NULL` to catch non-numeric values. **Example:** A Credit Card CSV has amount `"1,234.56"` with a comma – the system auto-fixes by removing the comma, but if a value is non-numeric like `"N/A"`, it triggers an ERROR.

- **INGEST-003: Duplicate Entry Detection** – **Description:** Checks for duplicate records in the raw data batch (potential double entries of the same transaction). **Severity:** WARN (duplicates are ingested but flagged for review). **Auto-fix:** N (cannot delete automatically without confirmation). **Owner:** Ingestion/Database Viewer. **Validation:** e.g. `SELECT transaction_hash, COUNT(*) c FROM staging GROUP BY transaction_hash HAVING c > 1` where `transaction_hash` is a SHA or

concatenation of key fields (date, amount, description) to identify duplicates. **Example:** Two identical expense entries appear in the Expenses list upload. The rule logs a WARN with evidence (the duplicate IDs) so the user or AI can later merge or remove one.

- **INGEST-004: Out-of-Bounds Date** – **Description:** Verifies transaction dates are within an expected range (e.g., not future dates beyond today, not ridiculously old). **Severity:** WARN (outliers flagged) or ERROR if far out of range. **Auto-fix:** N (requires user confirmation to adjust dates). **Owner:** Ingestion. **Validation:** `SELECT * FROM staging WHERE date > CURRENT_DATE + INTERVAL '1 day' OR date < '2000-01-01'`. **Example:** A Revenue entry has a date of 3025-01-01 due to a typo. The system flags this as an ERROR (invalid date) to prevent it from contaminating timeline analyses.

- **INGEST-005: Schema Evolution Check** – **Description:** Monitors incoming data for unexpected schema changes (extra columns, missing columns). **Severity:** ERROR (mismatched schema may indicate upstream source change). **Auto-fix:** N (manual intervention needed to handle schema change). **Owner:** Ingestion. **Validation:** The ingestion service compares the received schema to the expected schema definition (e.g., using a JSON schema or Data Contract). If a discrepancy is found (column count or names differ), it fails the job. **Example:** The Bank module's API starts sending a new field "`transaction_type`" that isn't in the contract – this rule catches it and halts ingestion so developers can update the schema contract accordingly.

Normalization & Transformation Rules

- **NORM-001: Valid Category Mapping** – **Description:** Ensures every transaction that should be categorized has a valid category after running the Category Mapper. No transactions remain uncategorized or mapped to an "Unknown" placeholder beyond a small threshold. **Severity:** WARN if a small percentage (e.g., <5%) are uncategorized, ERROR if higher. **Auto-fix:** Y (the AI Layer can auto-suggest categories for uncategorized items). **Owner:** Category Mapper module. **Validation:** e.g. `SELECT COUNT(*) FROM transactions_normalized WHERE category = 'Uncategorized'` and divide by total count. If >5%, raise ERROR; if >0, WARN. **Example:** After normalization, 2 out of 1000 transactions have category "Uncategorized" – this triggers a WARN (with those transaction IDs listed as evidence) and the AI suggests likely categories for them.

- **NORM-002: No Negative Values (Invalid Context)** – **Description:** Checks that certain fields are not negative unless logically expected. For instance, **Revenue** amounts should not be negative (expenses should not be positive, etc.), and **Budget** allocations shouldn't be negative. **Severity:** ERROR. **Auto-fix:** N (requires data correction at source). **Owner:** Revenue & Expenses modules. **Validation:** `SELECT * FROM revenue_normalized WHERE amount < 0` (and similar for expenses > 0, budgets < 0). **Example:** A user accidentally entered a negative revenue amount (perhaps trying to represent a refund). The rule catches this as an ERROR – revenue should be recorded as positive income or handled via a separate adjustment mechanism.

- **NORM-003: Referential Integrity** – **Description:** Ensures that references between modules are consistent. For example, every transaction in **Lists** (if Lists module tracks payees or entities) should reference a valid payee/category in the master lists; every category ID in transactions exists in the Category list. **Severity:** ERROR (data inconsistency). **Auto-fix:** N (cannot guess missing reference). **Owner:** Lists/Category Mapper. **Validation:** e.g.

`SELECT t.id FROM transactions_normalized t LEFT JOIN categories c ON t.category_id = c.id WHERE c.id IS NULL` – no rows should return. **Example:** If an **Expenses** entry references a category ID that doesn't exist (perhaps category was deleted or not synced), the rule fails, preventing that record from moving forward until the category reference is resolved or remapped.

- **NORM-004: Consistent Account Balances (Post-Normalization)** – **Description:** Verifies that for financial accounts (Bank, Credit Card), the ending balance calculated from transactions matches the ending balance provided (if any). Typically, Beginning Balance + Sum(transactions) = Ending Balance, within tolerance. **Severity:** WARN if slight discrepancy (e.g., <=1.5% of total), ERROR if large. **Auto-fix:** N (user must reconcile). **Owner:** Bank & Credit Card modules. **Validation:** After importing a statement's transactions, compute running balance and compare to stated ending balance. If diff > threshold or any unexplained gap in running balance, flag it. **Example:** A Credit Card statement says ending balance $1000, but summing the transactions and starting balance yields $990 – a 1% difference. This triggers a WARN (logged in `dq_results` with difference $10). The discrepancy might be due to a missing transaction or timing, and the user is alerted to investigate.

- **NORM-005: One-to-One Integrity between Modules** – **Description:** For certain cross-module data, ensure one source of truth. E.g., if **Bank** module marks an account as closed, ensure no new transactions come in that account; if an **Expense** is marked as reimbursed in **Lists** module, it should reflect in **Revenue** (as income) or a linked record. **Severity:** WARN (inconsistency flagged). **Auto-fix:** N. **Owner:** Depends on module (cross-functional). **Validation:** Varies by case. For account closure: `SELECT * FROM transactions_normalized WHERE account_status='closed' AND date > account_close_date`. **Example:** A bank account was closed in June, but an import shows a July transaction for that account – rule warns that data might be outdated or mis-labeled.

Account Balance & Reconciliation Rules

- **RECON-001: Bank vs Book Reconciliation** – **Description:** Compares bank account balances/transactions with the app's recorded **Revenue/Expenses** to ensure they match up. For each account and period, total inflows (revenues) minus outflows (expenses) in the app should equal the net change in bank balance (within tolerance ~1.5%). **Severity:** WARN if discrepancy <=1.5%, ERROR if larger. **Auto-fix:** N (needs investigation or manual adjustment). **Owner:** Finance/Bank module. **Validation:** The Reconciliation Engine links bank transactions to internal entries: for each account-month, `ABS((Sum(app_revenue) - Sum(app_expenses)) - (ending_balance - starting_balance)) / starting_balance`. If the percentage > 0.015, flag. Also check transaction-by-transaction matching where possible. **Example:** In September, the app records $10,000 revenue and $8,000 expenses for a bank account, net +$2,000, but the bank statement shows the balance only increased by $1,950. This 2.5% shortfall triggers an ERROR – perhaps some cash transaction wasn't recorded in the app. A reconciliation link is stored pointing out this account and period, with a difference of $50.

- **RECON-002: Cross-Account Transfer Consistency** – **Description:** Ensures that transfers between accounts are recorded on both sides. If the user transfers $500 from Bank A to Bank B, there should be a withdrawal in A and a deposit in B for the same date/amount. **Severity:** ERROR if missing on either side. **Auto-fix:** N (cannot generate missing transaction without confirmation). **Owner:** Bank module. **Validation:** When a transaction is categorized as "Transfer" (perhaps via Category Mapper

rules), the Recon Engine searches the other account for a matching counterpart. If none found, error. **Example:** A $500 transfer out of Checking on Oct 10 is in data, but no corresponding transfer into Savings on that date – flagged for user to add/correct the missing entry.

- **RECON-003: Credit Card Payment Match** – **Description:** Checks that credit card payments recorded in the Credit Card module match withdrawals in the Bank module. **Severity:** WARN (if timing differences, give a warning), ERROR if completely missing. **Auto-fix:** N. **Owner:** Credit Card/Bank. **Validation:** For each credit card payment (say, a transaction in Credit Card with type "Payment" of amount X on date D), look for a bank transaction (checking account) of amount X (within +/- 1 day of D). If not found or amount differs, flag. **Example:** User marked their credit card bill of $1000 as paid on Oct 5 (in Credit Card records), but no bank withdrawal ~Oct 5 for $1000 exists – likely the user hasn't imported that or mislabeled it. A WARN is logged so the user can reconcile this.

- **RECON-004: Duplicate Transaction Across Sources** – **Description:** Flags potential duplicate records across modules (e.g., the same expense imported twice via two different sources). For instance, an expense might appear in both a bank feed and a manual entry. **Severity:** WARN. **Auto-fix:** Y (the system could auto-merge or mark duplicates if high confidence). **Owner:** Expenses/Bank. **Validation:** After merging data from all sources, query for transactions with same date, amount, and payee appearing in multiple sources within a few days. **Example:** A $50 Grocery expense was recorded manually in **Expenses** and also came through in the **Bank** transactions import – the rule detects identical amount/date/vendor and flags one as a likely duplicate.

Categorization & Classification Rules

- **CATEG-001: Category Drift Detection** – **Description:** Monitors the distribution of transaction categories over time. If the category distribution shifts significantly compared to historical patterns, flag it (it could indicate mis-categorization or a genuine change in spending patterns). Uses Population Stability Index (PSI) to quantify drift [9]. **Severity:** WARN if PSI $\geq$ 0.2 (significant change [5]). **Auto-fix:** N (can't auto-change, but prompts review). **Owner:** Category Mapper/AI. **Validation:** Compute PSI between last month's category distribution and a prior baseline (e.g., same month last year or a rolling 3-month average) [10]. If PSI >= 0.2, log a WARN. **Example:** Normally the user spends 30% on Food, 20% on Rent, 10% on Utilities, etc., but this month a huge one-time purchase in a new category skews this (PSI = 0.25). The system raises a WARN that category distribution drifted significantly, highlighting the new category's contribution. (This might be okay, but alerts the user to confirm the categorization of that big purchase is correct).

- **CATEG-002: New Category Approval** – **Description:** If a transaction gets assigned to a category that is new or not in the approved list (perhaps via the AI categorizer), flag for review. **Severity:** WARN. **Auto-fix:** N (requires user confirmation). **Owner:** Category Mapper. **Validation:** `SELECT * FROM transactions_normalized t LEFT JOIN category_list cl ON t.category = cl.name WHERE cl.name IS NULL`. Any transaction with a category not in `category_list` yields a warning. **Example:** The AI Layer auto-categorizes a new expense as "Home Office". If "Home Office" wasn't a predefined category, the rule logs a WARN so the user can decide to add it as a new category or remap it.

- **CATEG-003: Consistent Categorization Rules** – **Description:** Ensures that similar transactions are categorized consistently. For instance, the same merchant (Amazon) should not flip between

"Shopping" and "Subscriptions" categories unless attributes differ. **Severity:** WARN. **Auto-fix:** Y (the AI could recategorize outliers). **Owner:** Category Mapper/AI. **Validation:** Group transactions by payee/description and check category variance: `SELECT payee, COUNT(DISTINCT category) FROM transactions_normalized GROUP BY payee HAVING COUNT(DISTINCT category) > 1`. If the same payee appears under multiple categories frequently, flag. **Example:** If "Starbucks" transactions appear sometimes as "Food" and sometimes "Office Expense", the rule warns of inconsistent categorization – suggesting the user standardize it. The AI might auto-fix by choosing the most frequent category or asking the user.

- **CATEG-004: Category Budget Alignment** – **Description:** (Related to Budget module) If a transaction's category is not part of any budget (and budgets are supposed to cover all spending categories), flag it. **Severity:** WARN. **Auto-fix:** N (user needs to adjust budget or category). **Owner:** Budget/Category Mapper. **Validation:** `SELECT DISTINCT category FROM transactions_normalized EXCEPT SELECT category FROM budget_allocations WHERE user_id = ...` to find categories with spend but no budget. **Example:** User spent $500 on "Pets" but never created a Pets budget category – the rule prompts either adding a budget for Pets or recategorizing those expenses under an existing budget category.

Budget & Financial Logic Rules

- **BUDGET-001: Budget Sum Consistency** – **Description:** Ensures the sum of category budgets equals the overall budget or parent budgets (if hierarchical). For a given user and period, the total allocated budget across categories should equal the total budget set in the **Dashboard** or master budget. **Severity:** WARN if slight mismatch (rounding), ERROR if major. **Auto-fix:** N (user must reconcile budgets). **Owner:** Budget module. **Validation:** `SELECT ABS(SUM(category_budget) - total_budget) as diff FROM budget_allocations` – diff should be 0 (or very low). Also enforce no category's budget exceeds some logical portion of total (e.g., cannot have category budget larger than total). **Example:** The user's monthly budget total is $5000, but the categories listed add up to $4800 – a $200 shortfall. The rule issues a WARN that $200 is unallocated (or missing) in the budget plan.

- **BUDGET-002: Overspend Alert** – **Description:** Checks if expenses in any category exceed that category's budget (or if total expenses exceed total budget) by more than a tolerance. **Severity:** WARN at 100%+ usage, ERROR if significantly over (e.g., >110% of budget). **Auto-fix:** N (cannot auto cut spending... but could auto-adjust budget target). **Owner:** Budget module. **Validation:** For each category: `SELECT category, SUM(expense_amount) / budget_amount AS utilization FROM budget_vs_actual WHERE period = current_month`. If utilization > 1 -> WARN, if >1.1 -> ERROR. **Example:** The user budgeted $300 for dining out, but has spent $330 (110%). This triggers an ERROR indicating the budget is exceeded. The `dq_results` entry might include evidence "Dining category at 110% of budget" and the Incident Bot could prompt the user to adjust the budget or curb spending.

- **BUDGET-003: Zero or Negative Budget Check** – **Description:** Ensures budgets are positive amounts and categories expected to have budgets do have them. **Severity:** WARN. **Auto-fix:** Y (for zero budgets, could auto-fill a nominal value or prompt user). **Owner:** Budget. **Validation:** `SELECT * FROM budget_allocations WHERE amount <= 0`. **Example:** If a user creates a

budget category but sets $0 (maybe intending to not spend at all or forgot to set), the system warns that a zero budget is set (which might be intentional, but flagged in case it's an omission).

- **BUDGET-004: Forecast vs Budget Alignment** – **Description:** Compares forecasted spending/income with budgets to see if any major divergence is predicted. If the AI **Forecast** for a category indicates the user will exceed the budget by, say, >10%, warn early. **Severity:** WARN. **Auto-fix:** N (user should adjust budget or note the variance). **Owner:** Budget/Forecast. **Validation:** Join forecast_projection with budget table: `SELECT category, forecast_amount, budget_amount FROM projections p JOIN budget b ON p.category = b.category WHERE forecast_period = next_month`. If `forecast_amount > 1.1 * budget_amount`, flag. **Example:** The model predicts $550 on groceries next month, but the budget is $500. This rule generates a WARN so the user is aware of a likely budget overshoot in advance.

Forecast & AI Layer Rules

- **FORECAST-001: Forecast Accuracy (SMAPE)** – **Description:** Evaluates forecasting model accuracy by Symmetric MAPE on recent predictions vs actuals. If SMAPE exceeds threshold (e.g. 10%), it indicates poor forecast performance [11]. **Severity:** WARN at >10%, ERROR if >20% (very inaccurate forecast). **Auto-fix:** N (but triggers model review/retrain suggestion). **Owner:** Forecast module / AI team. **Validation:** Compute SMAPE for the latest period: `SMAPE = 100/n * Σ(|Forecast - Actual| / ((|Actual|+|Forecast|)/2))`. The system records this in `forecast_metrics`. If SMAPE >= 10%, log a WARN; if >=20%, ERROR. **Example:** The app forecasted total expenses of $1000 for September, but actual was $1200 (SMAPE ≈ 18.2%). This is above 10%, so a WARN appears: "Forecast error 18% for Sep – exceeds 10% target." The AI Layer might automatically schedule a model retraining because we crossed the 0.2 PSI/error threshold that often signals the model is no longer reliable [6].

- **FORECAST-002: Forecast Freshness** – **Description:** Ensures forecasts are updated with the latest data (no stale predictions). For example, if it's now October and still showing a forecast generated in July for October, that's stale. **Severity:** WARN. **Auto-fix:** Y (the AI can auto-run an update if stale). **Owner:** Forecast/Intelligence Layer. **Validation:** Each forecast projection has a generation timestamp. Rule: if `current_date - forecast_last_updated > X` (e.g., >30 days or past period end), flag. **Example:** The user last ran the cash flow forecast 2 months ago; data has changed since. The system flags "Forecast is out-of-date" and automatically runs a new forecast job (auto-fix) to refresh it.

- **FORECAST-003: Anomaly Detection on Trends** – **Description:** Flags anomalies in time-series trends for key indicators (could be part of AI layer): e.g., a sudden 300% spike in expenses from one week to next with no known reason. **Severity:** WARN. **Auto-fix:** N (can't fix data unless it's an error). **Owner:** Dashboard/AI. **Validation:** Use statistical models or Z-score on week-over-week changes of aggregates. If change > threshold (like beyond 3σ or a configured percentage) and no corresponding increase in another module (like no large one-time income to explain huge expense spike), then flag. **Example:** Dashboard shows revenue dropped 80% this month vs last – the rule triggers a WARN for unusual drop, possibly indicating missing data (maybe some revenue entries weren't imported).

- **AI-001: Model Input Drift** – **Description:** (Forward-looking) Monitors the distribution of model input features for drift over time. If the input data to the AI models (e.g., transaction patterns for

forecasting, or spending behavior for categorization) drifts from the training data distribution, alert to possible model performance degradation. **Severity:** WARN. **Auto-fix:** N (but signals retraining needed). **Owner:** AI/Intelligence Layer. **Validation:** Compute drift metrics like PSI or KL divergence on key features vs training baseline. If PSI $\geq$ 0.2 on any feature, log a WARN (model drift) [5]. **Example:** The forecasting model was trained on a stable economy, but suddenly the user's transaction patterns during holidays differ greatly (PSI on daily expense amount distribution = 0.25). The system flags model input drift – the model might need retraining with holiday data. This ties into **model drift monitoring** to maintain AI accuracy.

Indicator Freshness & Coverage Rules

- **DATA-001: Data Freshness SLA** – **Description:** Checks that data from external sources (bank feeds, etc.) are up-to-date. For example, bank transactions should be refreshed daily. If no new data has been ingested beyond a threshold, raise an alert. **Severity:** WARN. **Auto-fix:** N (user might need to manually trigger sync or check connections). **Owner:** Bank/Credit Card modules. **Validation:** For each data source, track last successful ingest timestamp in `provenance`. If `NOW - last_ingest > SLA` (say 24 hours for daily data) then warn. **Example:** If the bank feed hasn't pulled any new transactions in 3 days (maybe API credentials expired), the rule warns "Bank data stale as of 3 days ago" so the user can re-link the account.

- **DATA-002: Heat Map Coverage** – **Description:** Ensures the Heat Map module has data for all expected combinations (e.g., all months vs categories). If the heat map expects a full year of monthly data, ensure each month has data points (even zero values) and every key category is represented (unless truly no data). **Severity:** WARN if gaps. **Auto-fix:** Y (system can insert zero placeholders if needed). **Owner:** Heat Map/Charts. **Validation:** Suppose the heat map projection table has one row per (month, category) with an amount. The rule checks for missing months or categories: e.g., if a month is missing entirely, or a category that has data in other months is missing in one month (implying possibly data drop or missing import). **Example:** The heat map for 2025 is missing data for March for all categories (perhaps a file wasn't loaded). The rule flags this gap. The system could auto-fix by inserting zero amounts for March or prompting the user to confirm if March had no activity or if data is missing.

- **DATA-003: Key Indicator Freshness** – **Description:** For any computed KPIs (Key Performance Indicators) shown on the Dashboard (like total balance, net worth, etc.), verify they are based on the latest available data. If any KPI hasn't been updated (e.g., due to a failed job) or seems stale compared to underlying data timestamps, flag it. **Severity:** WARN. **Auto-fix:** Y (attempt to recompute KPI). **Owner:** Dashboard/Intelligence Layer. **Validation:** Each KPI widget could store last computed time. Cross-check against last data update time in that domain. If data updated more recently than KPI, then KPI is stale. **Example:** The "Net Worth" figure on the Dashboard hasn't recalculated since last week despite new transactions yesterday – the rule catches that and refreshes the calculation automatically, so the user always sees fresh numbers.

Each rule above produces an entry in the `dq_results` table when violated (or passes can be logged for audit). The **Module Owner** field designates who should address the issue (useful if the system grows to multiple teams). Many WARNs, while not stopping the pipeline, will surface on the Dashboard's "Data Quality" panel or via the Incident Bot's alerts, so the user is aware of them. All rule violations include **contextual evidence** – e.g., sample record IDs, counts, or threshold values – to aid debugging. Crucially,

thanks to the lineage, any violation can be traced to the exact data contributing to it, **providing real traceability from symptom to source** [12].

Finally, rules with **Auto-fix = Y** will trigger automated remediation logic in the AI Layer. For instance, for **CATEG-001 (uncategorized transactions)**, the system will automatically attempt to categorize those transactions using the AI model and either directly fix them or provide suggestions. Auto-fixes are executed carefully (often for formatting issues or enrichment that doesn't risk data correctness), and any changes go through the Change-Set ledger for audit. Unresolved warnings feed into the change management process as described, ensuring continuous improvement of data quality.

Data Contracts & Schemas

To support the above architecture, the system defines clear data contracts and schemas for observability data. Below we provide both TypeScript type definitions (for use in application code or as API contracts) and SQL schema sketches for the key tables. All schemas are **multi-tenant ready** – they include a `userId` (or tenant identifier) to segregate data per user. They also include **context metadata** like `jobRunId` (to tie records to specific pipeline runs or DQ job executions) and fields to store **evidence** or references for traceability. Many tables have fields to support **status transitions** (e.g., `status` that can move from "NEW" to "RESOLVED") and to link derived data back to raw inputs (via IDs or foreign keys to lineage/provenance).

These schema definitions ensure that for every projection or summary data point, we can navigate back to source data and any DQ checks or lineage associated with it. They form the backbone of the observability layer's storage.

`dq_results` – Data Quality Check Results

This table logs the outcome of each data quality rule evaluation. Each row is a rule violation (or notable event) with details about severity, status, and lineage to source data. It is the primary table the Incident Bot watches and the AI references for context.

TypeScript interface:

```
interface DqResult {
  userId: string;            // Tenant or user scope
  resultId: string;          // Unique ID for this DQ result (e.g., UUID)
  ruleId: string;            // ID of the rule (e.g., "FORECAST-001")
  severity: 'WARN' | 'ERROR';
  status: 'NEW' | 'ACK' | 'RESOLVED' | 'IGNORED';
  module: string;            // Module or domain (e.g., 'Forecast', 'Bank')
  checkpoint: string;        // Pipeline stage (e.g., 'INGESTION', 'PROJECTION')
  message: string;           // Human-readable description of issue
  autoFixApplied: boolean;   // True if an auto-fix was executed
  evidence?: string;         // JSON or text with evidence (e.g., {"expected":
1000,"actual":950})
  jobRunId: string;          // ID of the pipeline/DQ run that produced this
```

```
  changeProposalId?: string; // ID of a change-set proposal if spawned
  createdAt: Date;
  resolvedAt?: Date;
}
```

SQL Schema:

```
CREATE TABLE dq_results (
  user_id        VARCHAR(64) NOT NULL,
  result_id      UUID NOT NULL PRIMARY KEY,
  rule_id        VARCHAR(32) NOT NULL,
  severity       VARCHAR(5) NOT NULL,     -- 'WARN' or 'ERROR'
  status         VARCHAR(10) NOT NULL,    -- 'NEW', 'ACK', 'RESOLVED', 'IGNORED'
  module         VARCHAR(50) NOT NULL,    -- e.g., 'Forecast', 'Budget'
  checkpoint     VARCHAR(20) NOT NULL,    -- e.g., 'INGESTION', 'NORMALIZATION'
  message        TEXT NOT NULL,           -- description of the issue
  auto_fix_applied BOOLEAN NOT NULL DEFAULT false,
  evidence       TEXT,                    -- details or JSON evidence of issue
  job_run_id     UUID NOT NULL,
  change_proposal_id UUID,               -- link to a change proposal if any
  created_at     TIMESTAMP NOT NULL DEFAULT NOW(),
  resolved_at    TIMESTAMP,
  -- Indexes for quick lookup by user, rule, status:
  UNIQUE(user_id, result_id),
  INDEX idx_dqresults_user_status (user_id, status),
  INDEX idx_dqresults_rule (user_id, rule_id),
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Notes: The primary key is a combination of `result_id` (globally unique) and implicitly `user_id` (to avoid any collision across tenants, we could also include user_id in PK). We store `rule_id` to know which rule triggered, `module` and `checkpoint` for context (these could also be derived from `rule_id` naming, but stored for convenience). The `status` field allows workflow: for example, when a WARN is reviewed and deemed okay, it might be marked "ACK" (acknowledged) or "IGNORED", whereas issues that get fixed would go to "RESOLVED". The table can accumulate history of issues; for trend analysis we might not hard-delete resolved ones. The `evidence` field is flexible, it might contain a snippet of offending data or a JSON object with multiple values (for example, actual vs expected values, or IDs of records that failed).

`change_proposal_id` links to an entry in a hypothetical change ledger (not detailed here) if the system created a change suggestion. This ties the DQ result to follow-up actions. The `job_run_id` links to the specific pipeline run – which in turn can be used to find logs or lineage specific to that run.

`lineage_edges` – Data Lineage Graph Edges

This table represents the directed relationships between data entities (or even individual records) as data transforms through the pipeline. Each row is an edge in the lineage DAG (Directed Acyclic Graph), capturing that a **downstream** entity/record was derived from an **upstream** entity/record.

TypeScript interface:

```
interface LineageEdge {
  userId: string;
  edgeId: string;                // Unique ID for this lineage edge
  upstreamEntity: string;        // e.g., 'raw.transactions'
  upstreamRecordId: string;      // e.g., ID of raw record (or batch ID)
  downstreamEntity: string;      // e.g., 'proj.monthly_summary'
  downstreamRecordId: string;    // ID of derived record (if applicable)
  relation: 'derived' | 'aggregated' | 'transformed';  // type of lineage link
  jobRunId: string;              // which job/run created this link
  evidence?: string;             // optional transformation details (e.g., SQL
query or expression)
  createdAt: Date;
}
```

SQL Schema:

```
CREATE TABLE lineage_edges (
  user_id           VARCHAR(64) NOT NULL,
  edge_id           UUID NOT NULL PRIMARY KEY,
  upstream_entity   VARCHAR(100) NOT NULL,   -- e.g., 'transactions_raw'
  upstream_record_id VARCHAR(100) NOT NULL,
-- could be a UUID or composite key of the record
  downstream_entity VARCHAR(100) NOT NULL,   -- e.g., 'monthly_projection'
  downstream_record_id VARCHAR(100) NOT NULL,
  relation          VARCHAR(20) NOT NULL,    -- e.g., 'derived', 'lookup',
'aggregated'
  job_run_id        UUID NOT NULL,
  evidence          TEXT,                    -- e.g., transformation formula or
query snippet
  created_at        TIMESTAMP NOT NULL DEFAULT NOW(),
  UNIQUE(user_id, edge_id),
  INDEX idx_lineage_downstream (user_id, downstream_entity,
downstream_record_id),
  INDEX idx_lineage_upstream (user_id, upstream_entity, upstream_record_id),
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Notes: This design can capture lineage at a *record level* granularity. For example, if monthly summary record `m789` is sum of transactions `tx123` and `tx124`, we would have two edges: (`tx123` -> `m789`) and (`tx124` -> `m789`). The `relation` field could have values like `'aggregated'` (many upstream to one downstream), `'derived'` (one-to-one transformation), or `'transformed'` (general case). The `evidence` field might store an expression or note (e.g., "m789.amount = SUM(tx.amount)" or a hash of transformation logic).

For efficiency, one might also store lineage at dataset level (like which table/view depends on which source tables), but here we focus on traceability down to record level for critical data. This table will be referenced when tracing DQ issues: e.g., given a problematic `downstream_record_id`, one can query `lineage_edges` to find all `upstream_record_id` that fed into it, then inspect those raw records via `provenance`. With the indexes provided, we can quickly retrieve lineage by either direction.

`provenance` – Data Provenance and Source Tracking

The provenance table records the origin details of raw data records (and possibly important derived records). It links internal record IDs to external source information and ingestion events. This is crucial for auditing and for providing evidence or drill-down for DQ issues.

TypeScript interface:

```
interface Provenance {
  userId: string;
  provId: string;            // Unique ID for this provenance record
  entity: string;            // Entity name, e.g., 'transactions_raw'
  recordId: string;          // Internal ID of the record in that entity
  sourceType: string;        // e.g., 'CSV File', 'Bank API', 'ManualEntry'
  sourceId: string;          // Identifier of source (file name, API endpoint,
etc.)
  sourceRecordId?:
string;   // If source data had an ID (e.g., primary key from source)
  ingestTime: Date;
  checksum:
string;          // Hash/fingerprint of the source record (for delta detection)
  jobRunId: string;          // The ingestion job that brought this record
}
```

SQL Schema:

```
CREATE TABLE provenance (
  user_id         VARCHAR(64) NOT NULL,
  prov_id         UUID NOT NULL PRIMARY KEY,
  entity          VARCHAR(100) NOT NULL,   -- e.g., 'transactions_raw'
  record_id       VARCHAR(100) NOT NULL,   -- internal record identifier
```

```
  source_type     VARCHAR(50) NOT NULL,    -- 'BankAPI', 'CSV', etc.
  source_id       VARCHAR(100) NOT NULL,   -- e.g., file name or API endpoint
  source_record_id VARCHAR(100),
-- if applicable, source's own record ID
  ingest_time     TIMESTAMP NOT NULL,
  checksum        VARCHAR(128) NOT NULL,   -- e.g., SHA-256 hash of the raw
record
  job_run_id      UUID NOT NULL,
  UNIQUE(user_id, entity, record_id),
  INDEX idx_prov_source (user_id, source_id),
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Notes: Each raw record gets a provenance entry when ingested. For instance, if a row is read from `Jan2025.csv`, we might have `source_type='CSV File'`, `source_id='Jan2025.csv'`, and possibly no `source_record_id` (or maybe the line number in file). For API, `source_id` might be an endpoint or account name, and `source_record_id` could be an external transaction ID. The `checksum` allows comparing if the same source record is ingested twice (for delta builds: if a file's content hasn't changed, re-import can be skipped). If a projection or derived table needs provenance (less common), we could also log those – but largely lineage covers derived records.

The combination of `provenance` and `lineage_edges` gives a full path: e.g., a DQ issue on a projection record -> lineage_edges tells which raw records contributed -> provenance of those raw records tells exactly which file or source they came from, and when. This fulfills the requirement that projections link back to raw inputs and evidence. For example, an unresolved WARN on a projection can be accompanied by the provenance of the source data that likely caused it (like "source_id: `Jan2025.csv`, line 42 had an outlier").

`reconciliation_links` – Reconciliation Pairings and Differences

This table captures the links between records that should represent the same real-world event or balance, typically used for reconciliation between two sources. Each row links, for example, a Bank transaction and an Expense record that are considered the same, and notes any discrepancy.

TypeScript interface:

```
interface ReconciliationLink {
  userId: string;
  linkId: string;
  sourceA: string;           // e.g., 'bank_transactions'
  recordAId: string;
  sourceB: string;           // e.g., 'expenses'
  recordBId: string;
  linkType: string;          // e.g., 'transaction_match', 'account_balance'
  status: 'MATCHED' | 'UNMATCHED' | 'PENDING';
  differenceAmount?: number; // numeric difference if any
```

```
  differencePct?: number;    // percentage difference if applicable
  evidence?: string;         // details, e.g., "Expected $1000, Found $990"
  jobRunId: string;
  createdAt: Date;
  resolvedAt?: Date;
}
```

SQL Schema:

```
CREATE TABLE reconciliation_links (
  user_id        VARCHAR(64) NOT NULL,
  link_id        UUID NOT NULL PRIMARY KEY,
  source_a       VARCHAR(50) NOT NULL,   -- e.g., 'bank_transactions'
  record_a_id    VARCHAR(100) NOT NULL,
  source_b       VARCHAR(50) NOT NULL,   -- e.g., 'expenses'
  record_b_id    VARCHAR(100) NOT NULL,
  link_type      VARCHAR(50) NOT NULL,   -- 'transaction_match' or
'balance_compare'
  status         VARCHAR(20) NOT NULL,   -- 'MATCHED', 'UNMATCHED', 'PENDING'
  difference_amount NUMERIC(18,2),       -- e.g., amount diff for balances
  difference_pct   DECIMAL(5,2),
  evidence       TEXT,
  job_run_id     UUID NOT NULL,
  created_at     TIMESTAMP NOT NULL DEFAULT NOW(),
  resolved_at    TIMESTAMP,
  UNIQUE(user_id, link_id),
  INDEX idx_recon_status (user_id, status),
  INDEX idx_recon_recordA (user_id, source_a, record_a_id),
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Notes: This table is populated by the Recon Engine. For direct transaction matching, `source_a` and `source_b` could be specific (like 'bank_tx' and 'credit_card_tx' for matching a payment). For account balance reconciliation, the link might not be record-to-record but rather summary-to-summary; in that case, `recordAId`/`recordBId` might point to an aggregate or account ID and `link_type='account_balance'`, with `difference_amount` indicating the balance difference. We include both `differenceAmount` and `differencePct` if applicable (for balances relative to expected).

The `status` can start as 'UNMATCHED' when a discrepancy is found. If the user resolves it (maybe by adding a missing transaction), the system can update status to 'MATCHED' and set `resolved_at`. Unresolved ones remain 'UNMATCHED' and could generate persistent WARNs. The `evidence` field can store a brief explanation (like which period or account this pertains to). By querying this table, the system (or user) can see all reconciliation issues at a glance.

This table also indirectly links to DQ results: e.g., RECON-001 rule might produce a DQ WARN with evidence "Account X off by $50 (linkId abc123)". The `reconciliation_links` entry with `link_id='abc123'` stores the details of that mismatch, and once resolved, the DQ issue can be marked resolved too. This separation of DQ logging (dq_results) and the detailed reconciliation data (reconciliation_links) allows flexible analysis (the latter is more granular data, the former is high-level alerts).

`forecast_metrics` – Forecast Performance Metrics

This table stores metrics related to forecasts, particularly error measurements, to track model performance over time. Each row could correspond to a forecast run or a forecast period's evaluation.

TypeScript interface:

```
interface ForecastMetric {
  userId: string;
  forecastId: string;       // ID of the forecast model or run
  period: string;           // The time period of forecast (e.g., '2025-09')
  horizon: string;          // How far ahead the forecast was made (e.g., '1
month ahead')
  actualValue: number;
  forecastValue: number;
  smape: number;            // Symmetric MAPE for this period's forecast
  mape?: number;
  mae?: number;
  createdAt: Date;
  jobRunId: string;
}
```

SQL Schema:

```
CREATE TABLE forecast_metrics (
  user_id       VARCHAR(64) NOT NULL,
  forecast_id   VARCHAR(64) NOT NULL,   -- identifier for the forecast model or
scenario
  period        VARCHAR(20) NOT NULL,   -- e.g., '2025-09'
  horizon       VARCHAR(20) NOT NULL,   -- e.g., '1_month_ahead'
  actual_value  NUMERIC(18,2),
  forecast_value NUMERIC(18,2),
  smape         DECIMAL(5,2),          -- e.g., 8.55 for 8.55%
  mape          DECIMAL(5,2),
  mae           NUMERIC(18,2),
  job_run_id    UUID NOT NULL,
  created_at    TIMESTAMP NOT NULL DEFAULT NOW(),
  PRIMARY KEY(user_id, forecast_id, period, horizon),
```

```
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Notes: We include common forecast error metrics like SMAPE (primary one used in rules), MAPE, MAE (Mean Absolute Error) etc., as columns. Each forecast run might generate multiple rows here if it predicts multiple periods; or we might store one row per period once actuals are available to compare. The primary key is composite including `period` and `horizon` to allow multiple forecast horizons (like one-month ahead vs three-months ahead forecasts).

This table is populated when actuals for a forecast period are known (e.g., at period end, compare predicted vs actual). The DQ rule FORECAST-001 uses this table: e.g., it might query the latest row for each period and check `smape`. We can retain history to see if accuracy is degrading over time (model drift).

An example entry: `{user_id: 42, forecast_id: 'main_v1', period: '2025-09', horizon: '1_month', actual_value:1200, forecast_value:1000, smape:18.2}` as in the earlier example. That would violate our threshold and correspond to a dq_result entry. Having the metric stored here means we can also display it in a model performance dashboard or trigger automatic retraining when certain criteria are met.

`categorization_metrics` – Categorization Quality Metrics

This table tracks metrics around the categorization of transactions and related data quality aspects (distribution drift, coverage of categories, etc.). It helps quantify how well our category mapping is performing.

TypeScript interface:

```
interface CategorizationMetric {
  userId: string;
  metricDate: Date;             // Date or period of the metric evaluation
  baselinePeriod?: string;      // Reference period for drift (if applicable)
  totalTransactions: number;
  uncategorizedCount: number;
  uncategorizedPct: number;
  distinctCategories: number;
  psiCategoryDrift?: number;    // PSI comparing category distribution to
baseline
  jobRunId: string;
}
```

SQL Schema:

```
CREATE TABLE categorization_metrics (
  user_id            VARCHAR(64) NOT NULL,
```

```
  metric_date        DATE NOT NULL,     -- e.g., '2025-09-30' for end-of-month
metrics
  baseline_period    VARCHAR(20),       -- e.g., '2025-08' if comparing Sep to
Aug
  total_transactions INT NOT NULL,
  uncategorized_count INT NOT NULL,
  uncategorized_pct  DECIMAL(5,2) NOT NULL,
  distinct_categories INT NOT NULL,
  psi_category_drift DECIMAL(5,3),      -- e.g., 0.134 = modest drift, 0.25 =
significant
  job_run_id         UUID NOT NULL,
  PRIMARY KEY(user_id, metric_date),
  FOREIGN KEY (user_id) REFERENCES users(id)
);
```

Notes: This table is updated perhaps monthly or with each run of the categorization checks. It provides a summary of categorization quality: - `uncategorized_pct` directly tells us how many transactions are left without category (should be near 0 ideally). - `distinct_categories` can track if new categories are being introduced frequently. - `psi_category_drift` is the metric for distribution change. For instance, if `psi_category_drift = 0.25` for September vs August, that is flagged as significant drift (since ≥0.2 is significant [5]).

The **baseline_period** indicates to which prior period the PSI was compared (could be previous month or a rolling average). Alternatively, we might store a separate table of pairwise comparisons, but for simplicity, storing the latest drift vs last month is fine.

This table feeds rules like CATEG-001 and CATEG-002. The DQ Runner, after computing category distribution for the month, inserts a row here and then evaluates if `psi_category_drift` >= threshold to decide on a WARN. Storing the data allows trend analysis: e.g., the user or system can see that PSI has been creeping up each month, indicating behavior change or accumulating misclassifications.

For lineage: if needed, we could link this back to which categories contributed most to drift (that might be stored in the evidence of the dq_result instead, listing top changing categories). But the metrics here give an overall picture. Also, `uncategorized_count` being non-zero will trigger rule CATEG-001 (Warn if >0). The AI can use this to say "Out of 1000 transactions, 5 are uncategorized (0.5%)."

Finally, note that all these tables include `user_id` as the first column in primary keys and indexes, enforcing tenant isolation (e.g., queries will always filter by user_id). This design scales for SaaS: each user's data quality records and metrics are handled separately, and could even reside in separate schema or databases if needed (as seen in some architectures where each user gets their own dataset [3]).

Status Transitions & Trace Links: In practice, an issue entry in `dq_results` can be connected to the detailed data that caused it. For example, a DQ result for forecast error will have a `rule_id='FORECAST-001'` and the `message` might say "SMAPE 18.2% for 2025-09, expected < 10%". The `forecast_metrics` table will have the exact values for that period. Using `job_run_id` one could join or correlate the entries if needed (the job run that computed the metric and the one that generated the

DQ result would match). Similarly, a categorization drift issue will point to `categorization_metrics` for the value of PSI. And a reconciliation warning will reference a `reconciliation_links.link_id` where one can see the two records that didn't match.

By designing schemas with these reference IDs and context fields, we ensure **every DQ rule violation is traceable and auditable**. One can move from a high-level alert in `dq_results` to the supporting evidence in metrics tables, to the actual records via lineage/provenance, and even to the remediation via change proposals. The end result is a robust observability and data quality framework that not only detects and reports issues, but also ties them into the data's lineage and lifecycle, supporting automated healing and continuous improvement of the financial intelligence app's data accuracy and reliability.

Sources:

1. Bigeye Data Observability – on monitoring vs lineage for tracing issues [8] [2]
2. Matthew Burke's Blog – on Population Stability Index thresholds for distribution drift [5]
3. Vexpower (Taylor, 2022) – on interpreting forecast error percentages (MAPE) [11]
4. DQOps Documentation – on multi-tenant data quality storage (per-user datasets for results) [3]
5. Medium (Moronta, 2023) – example of lineage aiding traceability of data quality issues [12]

---

[1] [2] [7] [8] Monitoring vs. Lineage: Why You Need Both For Data Observability Success
https://www.bigeye.com/blog/monitoring-vs-lineage-why-you-need-both-for-data-observability-success

[3] [4] DQOps Data Quality Engine Architecture
https://dqops.com/docs/dqo-concepts/architecture/dqops-architecture/

[5] [6] [9] [10] Population Stability Index | Matthew Burke's Blog
https://mwburke.github.io/data%20science/2018/04/29/population-stability-index.html

[11] Learn Mean Absolute Percentage Error | Vexpower
https://www.vexpower.com/brief/mean-absolute-percentage-error

[12] Column-Level Lineage and Data Quality in dbt with SQLGlot and Elementary | by Sendoa Moronta | Towards Data Engineering | Medium
https://medium.com/towards-data-engineering/linaje-de-columna-y-calidad-de-datos-en-dbt-con-sqlglot-elementary-36f0c80b8421