

Unified Payment Intelligence Table - Architecture Strategy

Overview

This document outlines an approach for consolidating four separate payment data sources into a single unified table for fraud detection. The design takes into account that this would be a first MLOps implementation with a small team (2-3 people) and the reality that upstream systems can change without warning.

The main approach is a Medallion architecture (Bronze/Silver/Gold layers). Bronze keeps raw data as-is for when things break. Silver cleans and validates. Gold is the final ML-ready table.

Goal for first 6 months: Get the unified table up and running with good data quality checks, make life easier for data scientists, and lay groundwork for real-time capabilities if needed later.

Tech stack: Airflow, Spark, Parquet, PostgreSQL, Great Expectations. Kept it simple - this can be run by 1-2 people without needing a huge ops team. Streaming tech like Kafka/Flink can wait until there's an actual business need.

What This Document Delivers

Data Architecture Design : - **Real-time serving strategy:** Payment API streaming ingestion (15-min micro-batches), with path to full real-time in Month 7+ if needed (Section 4) - **Batch processing strategy:** Medallion architecture with Airflow orchestration, daily Gold table refresh, handles four sources with different timing (Section 2) - **Feature computation strategy:** Gold layer produces ML-ready unified table with point-in-time correctness to prevent data leakage (Section 1)

Data Quality & Monitoring Strategy: - **Three-layer validation:** Bronze (schema), Silver (business logic), Gold (feature quality) using Great Expectations (Section 3) - **Monitoring setup:** Prometheus + Grafana tracking freshness, validation rates, pipeline health with tiered alerting (Section 3) - **Upstream reliability handling:** Defensive design with schema fingerprinting, dead letter queue, continues processing even with bad data (Section 3) - **Schema change detection:** Automated detection and alerting when upstream systems change without notice (Section 3)

1. Unified Payment Intelligence Table: Schema Design

Schema

Here's the proposed unified table schema. It's denormalized to make life easier for data scientists - everything in one place rather than joining four tables:

```
CREATE TABLE gold.payment_intelligence_unified (  
  -- Primary identifiers  
  payment_id          STRING NOT NULL,  
  src_account_id      STRING NOT NULL,  
  dest_account_id     STRING NOT NULL,  
  payment_reference   STRING,  
  
  -- Transaction attributes (from Payment Transactions)  
  amount              DECIMAL(18,2) NOT NULL,  
  currency_code       STRING DEFAULT 'GBP',
```

```

payment_timestamp      TIMESTAMP NOT NULL,

-- Source account context (from Account Details)
src_account_opening_date    TIMESTAMP,
src_account_age_days        INTEGER,
src_account_status          STRING,

-- Destination account context (from Account Details)
dest_account_opening_date   TIMESTAMP,
dest_account_age_days        INTEGER,
dest_account_status          STRING,

-- External risk signals (from External Risk Feed)
src_risk_flag               INTEGER DEFAULT 0, -- 0=clean, 1=suspicious, 2=confirmed
src_risk_last_updated        TIMESTAMP,
dest_risk_flag               INTEGER DEFAULT 0,
dest_risk_last_updated        TIMESTAMP,

-- Fraud labels (from Historical Fraud Cases - delayed 30-60 days)
fraud_flag                  BOOLEAN,
fraud_type                   STRING,
fraud_reported_date          TIMESTAMP,
fraud_label_source           STRING, -- 'api', 'csv', 'excel_analyst'

-- Metadata for lineage & quality
bronze_ingestion_timestamp   TIMESTAMP NOT NULL,
silver_processing_timestamp   TIMESTAMP NOT NULL,
gold_feature_timestamp        TIMESTAMP NOT NULL,
data_quality_flags            ARRAY<STRING>, -- ['missing_account_details', 'stale_risk_data']
schema_version                STRING NOT NULL,

-- Partitioning
payment_date                  DATE NOT NULL -- derived from payment_timestamp
)
PARTITIONED BY (payment_date)
STORED AS PARQUET;

```

Key design choices: - Denormalized so data scientists don't have to join multiple tables - Missing risk flags default to 0 (assume clean unless proven otherwise) - Fraud labels stay NULL if not available (different from "not fraud") - Three timestamps (bronze/silver/gold) help with debugging when things break - Quality flags array lets us track issues without blocking the pipeline - payment_timestamp is when the transaction happened, not when we ingested it

Data Source Patterns

Each source has different timing:

1. **Payment Transactions:** Real-time API stream, process every 15 minutes
2. **Account Details:** Daily CSV batch that arrives overnight, run at 6 AM

3. **External Risk Feed:** Irregular batch loads, process when they show up
4. **Fraud Labels:** Delayed 30-60 days, comes from API and manual files

Important point: The `payment_timestamp` field is when the transaction actually happened, not when we got the data. This matters for training models - we need to make sure we're not using future information that wouldn't have been available at transaction time.

2. Architecture: Medallion Pattern

Why This Approach

Given the organizational context ("Engineering doesn't tell us anything and they make a change... our pipelines just crash"), the Medallion architecture makes sense. It gives us an immutable audit trail of what upstream systems actually sent, even if it's invalid or breaks our expectations.

Three layers:

Bronze Layer - Raw data storage - Keep data exactly as received, no transformations - Even invalid records get stored with error flags - Retain for 90 days minimum (covers fraud label delays) - Use Parquet files, partitioned by date/hour

Silver Layer - Cleaned and validated - Deduplicate, fix data types, standardize formats - Validate referential integrity (payments link to accounts) - Enrich with derived fields (account age, etc.) - Keep historical snapshots for point-in-time joins

Gold Layer - ML-ready unified table - Final denormalized table (the schema above) - Point-in-time correct for training - Used for batch training, eventually real-time scoring

Pipeline Tools

Orchestration: Airflow - DAG 1: `payment_api_ingestion` (every 15 min) - DAG 2: `account_csv_ingestion` (daily 6 AM) - DAG 3: `risk_feed_ingestion` (triggered by file arrival) - DAG 4: `fraud_label_ingestion` (API poll + file watch) - DAG 5: `silver_transformation` (every 4 hours) - DAG 6: `gold_unified_table` (daily 7 AM)

Processing: Apache Spark - Handles the actual transformations - Run in standalone mode initially (can scale later) - Bronze → Silver → Gold jobs triggered by Airflow

Storage: Parquet files - Bronze: `/bronze/{source}/date={date}/hour={hour}/.parquet` - Silver: `/silver/{cleaned_source}/date={date}/.parquet` - Gold: `/gold/payment_intelligence_unified/date={date}/*.parquet`

Validation: Great Expectations - At Bronze boundary: schema checks, required fields, row counts - At Silver: business logic (amounts > 0, valid dates, referential integrity) - At Gold: feature validation (`account_age` >= 0, distributions look reasonable) - Failures go to dead letter queue, don't block the pipeline

Supporting Infrastructure - PostgreSQL: Airflow metadata, validation results, schema registry, metrics - Prometheus + Grafana: Monitor DAG success rates, task durations, data quality metrics - MLflow (add later): Model registry and experiment tracking

How Each Source Flows Through

Payment Transactions (every 15 min): - Airflow calls the API, gets last 15 min of transactions - Validate schema and write to Bronze (raw JSON → Parquet) - Silver: deduplicate by `payment_id`,

parse timestamps, validate amounts - Gold: join to accounts and risk flags

Account Details (daily at 6 AM): - Airflow waits for CSV file arrival - Validate and write to Bronze - Silver: create dated snapshot, compute `account_age_days` - Gold: point-in-time join (use account details from transaction date, not current)

Risk Feed (irregular): - Airflow file sensor watches for new files - Validate `account_id` and `risk_flag` values - Bronze: store with arrival timestamp - Silver: deduplicate to latest `risk_flag` per account - Gold: left join with default `risk_flag=0` for missing accounts

Fraud Labels (delayed 30-60 days): - Airflow polls API hourly and watches for analyst files (CSV/Excel) - Validate `payment_id` and `fraud_type` - Bronze: append-only storage (keep all versions) - Silver: materialize latest label per `payment_id` - Gold: left join (NULL means unlabeled, not “not fraud”)

Point-in-Time Correctness

This is important for training models without data leakage. Fraud labels arrive 30-60 days late, and account details update daily. We need to make sure training data only uses information that was actually available at transaction time.

Approach: - Keep historical snapshots of account details (one per day) - When building the Gold table, join accounts AS OF the payment date, not today - For training, use fraud labels as of `training_date` (e.g., `T+60` for mature labels) - This way features only reflect info available when the transaction happened

Can implement this in Spark using temporal joins with window functions.

3. Data Quality & Monitoring

Validation Layers

Bronze Layer - Check: schema structure, required fields, row counts roughly within expected range, unique IDs - Tool: Great Expectations - On failure: Write to dead letter queue, alert Slack, but keep processing - Philosophy: Never lose data, even if it's bad

Silver Layer - Check: business logic (amounts > 0 , dates valid), referential integrity (joins work), completeness ($>95\%$ have account details) - Tool: Great Expectations + Pandera - On failure: If $>5\%$ failures, block Gold propagation and alert - Philosophy: Don't propagate garbage

Gold Layer - Check: feature ranges (`account_age` ≥ 0), distributions look reasonable, fraud coverage increasing over time - On failure: Flag table as degraded, alert data science team - Philosophy: Make sure ML data is actually usable

Monitoring

Use Prometheus + Grafana to track: - Data freshness (hours since last update) - Validation pass rates (target $>95\%$) - DAG success rates and execution times - Unified table record counts and quality flags

Alerting: - Critical (page someone): Bronze ingestion down >1 hour, validation $<90\%$ - Warning (Slack): Data stale >4 hours, quality declining - Info (daily digest): Slow jobs, schema changes detected

Schema Changes

Since upstream teams don't always communicate changes, handle this defensively:

At Bronze ingestion: - Compute schema fingerprint (hash of fields + types) - Compare to expected schema stored in Git - If mismatch: log to dead letter queue, alert, but keep processing - Never lose data because of schema issues

Schema registry: - Keep expected schemas in Git as Avro definitions - Version them (1.0.0 → 1.1.0 for additions, 2.0.0 for breaking) - Ask Engineering for 2+ weeks notice on changes (though they won't always do it)

4. Implementation Plan (6 Months)

Months 1-3: Get It Working

Week 1-2: Setup infrastructure - Deploy Airflow, Spark, PostgreSQL - Set up Parquet storage (HDFS or MinIO)

Week 3-6: Bronze layer - Get all four sources ingesting to Bronze - Add validation with Great Expectations - Set up dead letter queue

Week 7-12: Silver and basic Gold - Build Silver transformations (dedupe, clean, join) - Create historical account snapshots - Build basic Gold unified table - Set up Prometheus + Grafana monitoring

Goal: Data scientists can query the unified table by end of month 3

Months 4-6: Make It Good

Week 13-18: Better quality - Comprehensive validation suites - Daily quality reports to Slack - Schema registry in Git - Document common issues and fixes

Week 19-24: More features - Add more fraud features (velocity, patterns, etc.) - Optimize Spark jobs - Implement incremental processing - Add MLflow for model versioning

Goal: >95% validation pass rate, pipeline runs hands-off for weeks

Month 7+: Real-Time

Only add streaming if there's a real business need for sub-hour detection: - Deploy Kafka for payment stream - Streaming feature computation (Kafka Streams or Flink) - Real-time scoring API

Note: Streaming adds 5-10x operational complexity. Only do it if batch isn't good enough.

Technology Stack

Use These (Months 1-6)

- **Airflow:** Orchestration - standard choice, lots of connectors
- **Spark:** Processing - distributed, SQL interface, well understood
- **Parquet:** Storage - columnar, compressed, works everywhere
- **PostgreSQL:** Metadata - Airflow backend, validation results

- **Great Expectations:** Data quality - comprehensive validation
- **Prometheus + Grafana:** Monitoring - open source, proven
- **MLflow:** Model registry - simple, standard

Can be run by 1-2 people once set up.

Skip These (For Now)

- **Kafka:** Too much operational overhead until there's a real-time requirement
- **Flink:** Stateful streaming is hard to debug, need it after Kafka
- **Feast:** Feature store adds complexity, batch is fine initially
- **Delta Lake:** Time-travel is nice but Parquet works fine

Add these later if actual business needs justify the complexity. Keep it simple.

Dealing with Organizational Reality

Problem: Engineering doesn't communicate changes, pipelines break randomly **Approach:** Bronze layer keeps everything, even bad data. Schema validation logs issues but doesn't stop processing. Dead letter queue for forensics. Try to build relationships with bi-weekly syncs.

Problem: Data scattered everywhere, no single source of truth **Approach:** Make the Gold table the official source. Document it, train people on it, measure adoption.

Problem: Small team, limited time **Approach:** Only deploy what's necessary. 6-7 core tools, no bloat. Operational simplicity matters more than fancy features.

Problem: Need to show value quickly **Approach:** Months 1-3 should deliver visible wins - unified table people can query, validation catching real issues, dashboard showing pipeline status.

Success Metrics (Month 6)

Pipeline reliability: Running hands-off for 30+ days (vs hours currently) Data quality: >95% validation pass rate Processing time: <2 hours end-to-end for daily refresh Features available: 100+ in Gold table DS productivity: New features take days not weeks Collaboration: Engineering actually tells us about some changes

Wrap Up

This approach gets a unified payment table built with a small team in an environment where things break unexpectedly. Medallion architecture is defensive by design. Tech stack is simple enough for 1-2 people to run. Implementation is incremental - value every month, not an 18-month big bang.

The goal is to turn scattered data chaos into something reliable and usable without needing enterprise-scale resources.