Lakehouse Mini Project

Architecture & Design Document

Data Engineering Take-Home Task

Audicin

Nastaran Ebrahimi - February 2026

This document describes the design decisions behind a three-layer (Bronze/Silver/Gold) Lakehouse pipeline that processes **application events, subscription records, and marketing spend data** to produce eight analytics-ready gold tables. The stack is intentionally lightweight—**Python + Pandas + DuckDB + Parquet**—and the pipeline is fully idempotent: safe to re-run at any time without corrupting results.

1. Architecture Overview

The pipeline implements a classic **medallion architecture** with three explicit layers: Bronze, Silver, and Gold. Each layer has a single, well-defined responsibility, making the system easy to reason about, debug, and extend.

Layer	Location	Purpose	Format
Bronze	lakehouse/bronze/	Raw ingestion – parse only, never filter	Parquet (all-string columns)
Silver	lakehouse/silver/	Clean, validate, normalise, deduplicate	Parquet (typed columns)
Gold	lakehouse/gold/	Business-logic aggregations, analytic tables	Parquet + DuckDB views,tables
Quarantine	lakehouse/quarantine/	Corrupt / rejected rows isolated per layer	Parquet (for audit & replay)

1.1 Why a Separate Quarantine Zone?

Corrupted or unparseable rows are never silently dropped. They are written to a dedicated quarantine directory with their source line number and error message preserved. This makes data-quality monitoring straightforward—a simple row-count on the quarantine files serves as an early-warning signal—and enables replay once the root cause is fixed.

1.2 Data Flow

The pipeline is orchestrated as a linear, stage-gated sequence:

Raw files (NDJSON / JSON / CSV)	→	**Bronze (parse only)**	→	**Silver (clean + validate)**	→	**Gold (aggregate)**	→	**DuckDB views**

2. Storage Format Choices

Parquet was chosen as the intermediary format for all three layers because it offers columnar compression (significantly reducing file sizes for analytics workloads), schema enforcement at read time, and native type support that eliminates round-trip string serialisation overhead. It also enjoys first-class support from Pandas and DuckDB, which are the two main engines used here.

DuckDB serves as the final query layer. Instead of embedding analytical SQL logic inside Python, all Gold Parquet outputs are materialized into DuckDB tables within the analytics schema. This provides analysts with a single connection endpoint (gold.duckdb) where they can execute arbitrary SQL across all gold tables, including cross-table joins (e.g., LTV/CAC ratio), aggregations, and dimensional analysis.

Decision	Rationale
All-string Bronze	Preserves raw data exactly as received. Type coercions happen once in Silver, with failures quarantined.
Typed Silver Parquet	Pandas infers types from the now-clean data. DuckDB reads typed Parquet natively—no casting at query time.
DuckDB Gold tables	Gold Parquet outputs are materialized into DuckDB tables within the analytics schema. This decouples storage (Parquet) from the query engine, providing analysts with ANSI SQL access without requiring an external database server.

3. Messy Data Handling

The dataset contains ten categories of intentional data quality issues. Below is an explicit account of how each is detected and resolved.

3.1 Corrupted / Unparseable JSON Lines (Bronze Layer)

Each line of `events.ndjson` is parsed individually inside a `try/except` block. Lines that raise any exception are written to `quarantine/events_bad_rows.parquet` together with their source line number and the error message. Valid lines proceed to Bronze as-is.

3.2 Duplicate Events

Two flavours of duplicates exist in the dataset:

- 1) **Exact duplicates:** Rows sharing the same event_id and identical payload are resolved by simple deduplication on `event_id` in the Silver layer.
- 2) **Conflicting duplicates:** Rows sharing the same event_id but differing in payload (e.g., different amount or currency) are resolved using a deterministic last-write-wins rule. The retained row is determined as follows:
- The record with the most recent `event_ts_utc` is kept.
- If multiple rows share the same timestamp, the one with the higher `_source_line_no` is retained. This is implemented by sorting the dataset using:

```
(event_id ASC, event_ts_utc DESC, _source_line_no DESC)
```

```
drop_duplicates(subset=["event_id"], keep="first")
```

3.3 Out-of-Order & Late Events

Timestamp ordering in the source file is not assumed. All timestamp-based aggregations in Silver and Gold operate on the parsed UTC timestamp, not on file ordering. Refunds can therefore precede their corresponding purchase events in the file, and this is handled correctly because Gold computes signed amounts independently of chronological order.

3.4 Inconsistent Timestamp Formats

The Silver events layer uses `dateutil.parser.parse()` which handles ISO 8601 with Z suffix, ISO 8601 with numeric offset, and space-separated `YYYY-MM-DD HH:MM:SS` formats automatically. All parsed timestamps are converted to UTC-naive datetimes stored in `event_ts_utc`. Rows whose timestamp cannot be parsed are quarantined.

3.5 Schema Evolution (v1 vs v2)

schema_version 1 rows may lack `currency` and `tax` fields. Silver guarantees all expected columns exist by injecting them as `None` if absent:

```
for c in needed_columns:
    if c not in df.columns:
        df[c] = None
```

This forward-compatible schema pinning means that adding a schema_version 3 in future requires only extending the needed_columns list.

3.6 Missing / Null user_id

Events without a valid `user_id` are retained in Silver (they may be useful for funnel analysis) but are excluded from any metric that requires user attribution (DAU, LTV, CAC). The exclusion filter consistently checks for the string representations `''`, `'none'`, and `'nan'` in addition to actual null values, because Bronze casts everything to strings.

3.7 Negative Spend & Marketing Data Issues

- Negative spend rows are quarantined in Silver marketing processing (spend_num < 0).
- Duplicate channel/date rows are aggregated by summation (groupby date + channel).
- Missing days are filled with zero spend to produce a continuous date-channel grid, ensuring correct CAC denominators even on zero-spend days.

3.8 Subscription Edge Cases

- Duplicate subscription_id: resolved by keeping the most recent record (sort by created_at DESC, drop_duplicates on subscription_id).
- Overlapping subscriptions: detected by comparing each subscription's start_date against the previous subscription's effective end_date for the same user. Overlapping records are quarantined rather than arbitrarily merged.
- Reactivations: flagged with a boolean reactivated column and gap_days metric — useful for churn-rate analysis without being quarantined.

4. Incremental Processing & Idempotency

4.1 Current Behaviour

Each pipeline run performs a full refresh: it reads all source files, re-processes every record from scratch, and overwrites the output Parquet files. This is appropriate for the dataset sizes in this assessment and guarantees idempotency by construction—running the pipeline twice produces bit-for-bit identical output.

4.2 Path to Incremental Processing

For production scale, the natural extension is:

- Partition Bronze and Silver Parquet files by `event_date` (YYYY-MM-DD).
- Track the max processed timestamp in a lightweight checkpoint file (e.g. last_run.json).
- On each run, read only partitions where `event_date` >= checkpoint date.
- Merge the new partition into the Gold aggregation tables using DuckDB's INSERT OR REPLACE semantics or an Iceberg MERGE INTO statement.

4.3 Idempotency Under Partial Failures

Because each stage writes to a separate directory, a failure in the Gold stage leaves Bronze and Silver outputs intact. The stages can therefore be re-run independently.

The DuckDB registration step is also idempotent. Each table in the analytics schema is rebuilt using:

`DROP TABLE IF EXISTS`

followed by `CREATE TABLE AS SELECT * FROM read_parquet(...)`

This guarantees that re-running the loader fully replaces the previous version of each table, preventing duplicate data and ensuring consistent results across executions.

5. Backfill Strategy

Because the pipeline is a full-refresh by design, backfilling historical data is trivially achieved by re-running the pipeline with the original source files in place. No special backfill mode is required.

For the incremental variant described in 4.2, a backfill would:

- Set the checkpoint date to the desired start of the historical window.
- Delete the relevant output Parquet partitions.
- Run the pipeline—it will re-process only the affected date range.

This avoids the need to re-process the entire history while still producing correct, deduplicated results.

6. Gold Table Design Decisions

Gold Table	Key Decision	Explanation
daily_active_users	Active = login, page_view, purchase	Signup alone is not an 'active' session; it is a one-time acquisition event.
daily_revenue_gross	Sum of purchase amounts per date	Does not net off refunds. Negative purchase amounts are included (data trap explicitly not cleaned here so analysts can see gross before netting).
daily_revenue_net	purchase (+) + refund (-) per date	Refunds are booked on their own event date, not back-dated to the original purchase. This matches standard revenue-recognition practice.
mrr_daily	Active-subscription sum per calendar day	MRR is computed daily (not monthly) to capture intra-month churn and reactivation accurately. Each subscription contributes price to every day it is active.
weekly_cohort_retention	Week of first signup as cohort key	Activity = login, page_view, purchase, trial_start, trial_convert. Week index 0 = signup week. Retention = active_users / cohort_size.
cac_by_channel	CAC per acquisition channel	'Paid conversion' = first purchase event per user. Channel attribution from earliest signup event. Users without a signup are attributed to 'Unknown'.
ltv_per_user	Refund-adjusted cumulative value	LTV = sum of signed amounts (purchase positive, refund negative) per user across all time.
ltv_cac_ratio	Aggregate ratio	LTV:CAC = total_ltv / cac_overall. Computed once as a single-row summary table. NULL when cac_overall = 0.

7. Observability & Data Quality

7.1 Structured Logging

Every pipeline stage emits structured log lines via a shared `get_logger()` utility. Each step logs START / END / FAILED events with a `duration_s` measurement, the module name, and row counts. This gives a complete audit trail from a single log file, making SLA tracking straightforward.

7.2 Built-in Data Quality Signals

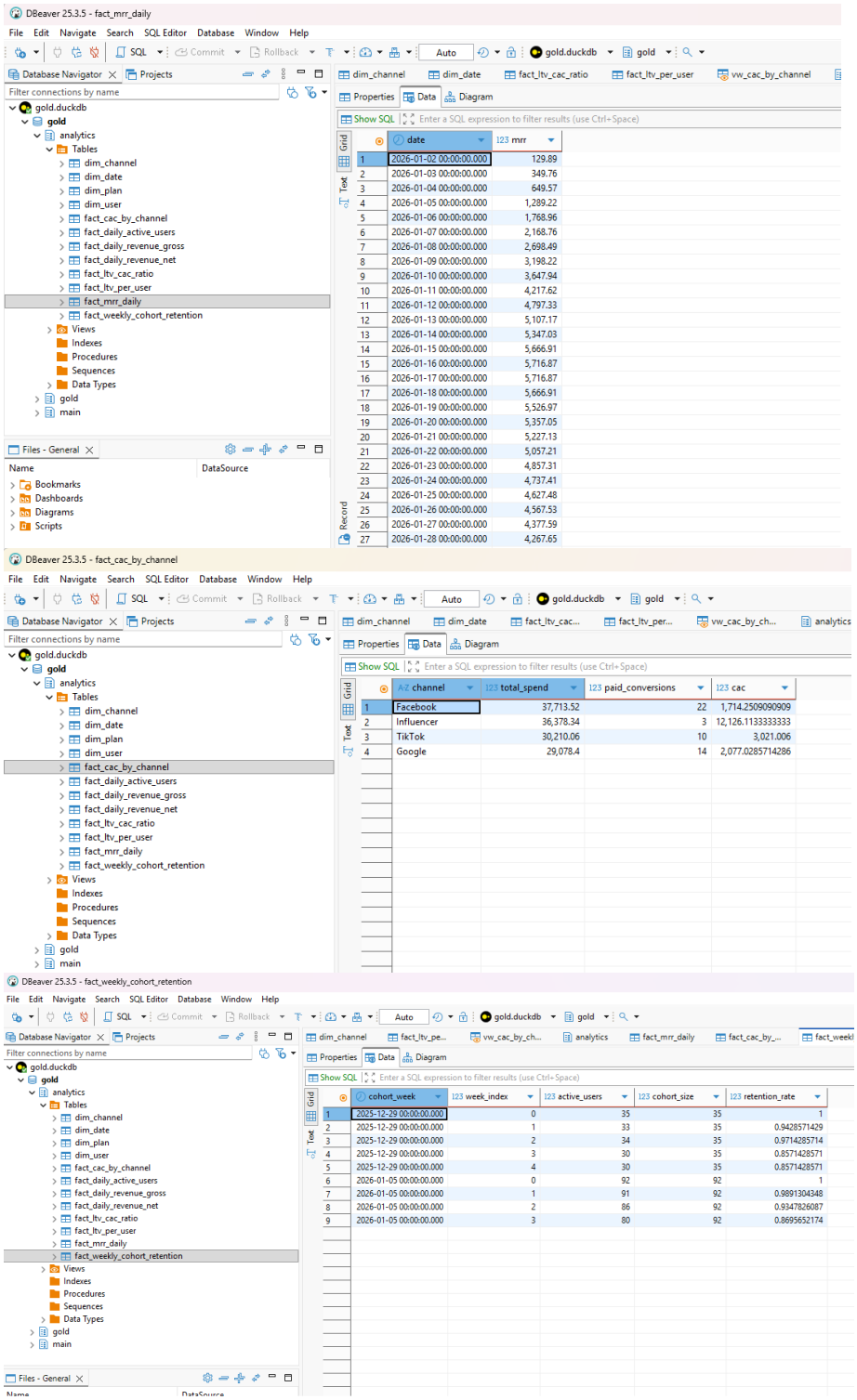
The following quality metrics are surfaced automatically on every run:

Signal	Where it appears
Quarantine row count	Logged per stage; quarantine Parquet files available for audit.
Duplicate event rate	Derivable: (bronze_count - silver_count) / bronze_count.
Late event detection	Refund refers_to_event_id is preserved; late-event detection is possible but not implemented.
Overlap subscription count	Logged explicitly in silver_subscriptions stage.
Reactivation count	Logged and flagged in Silver subscriptions output column.

7.3 Recommended Production Alerts

- Volume anomaly: daily event count deviates > 2 standard deviations from a 7-day rolling mean.
- Duplicate rate: quarantine_count / total_count > 5% triggers a WARNING.
- Freshness: latest event_ts_utc in Silver is older than 25 hours (for daily pipelines).
- Zero-revenue days: alert if daily_revenue_gross = 0 on a business day.

8. Gold Layer Analytics Summary



The Gold layer materializes core business KPIs in DuckDB under the analytics schema, enabling cross-table SQL analysis across revenue, retention, and acquisition performance.

Retention Analysis
Weekly cohort retention shows strong early engagement and gradual, healthy decay over time. Both visible cohorts retain approximately 85–90% of users by Week 3, indicating strong onboarding effectiveness and low early churn. Retention curves follow expected behavior, with no abnormal drop-offs.

Revenue & MRR Trend
Daily MRR demonstrates consistent upward growth over the observed period, increasing steadily without major volatility. This suggests net positive subscription growth and limited churn impact. The revenue base compounds smoothly over time, indicating stable monetization performance.

Customer Acquisition Cost (CAC)
Channel-level CAC reveals significant performance differences:

Facebook and Google show the most efficient acquisition costs.

TikTok performs moderately.

Influencer marketing has a substantially higher CAC, indicating lower efficiency and potential need for budget reallocation.

Overall Business Signal
The combined metrics indicate a healthy growth profile:

Strong retention

Increasing recurring revenue

Scalable acquisition channels

Clear visibility into cost efficiency by channel

The Gold layer successfully exposes actionable analytics through a structured fact/dimension model, enabling performance monitoring and strategic decision-making via SQL.

9. Project Structure

```
AudicinProject/
  data/                    # source files (events.ndjson, subscriptions.json,
marketing_spend.csv)
  lakehouse/
    bronze/                # raw Parquet (all-string)
    silver/                # typed, clean Parquet
    gold/                  # aggregated Parquet + analytics.duckdb
    quarantine/            # rejected rows with error context
  src/
    bronze/                # events.py  subscriptions.py  marketing.py
    silver/                # events.py  subscriptions.py  marketing.py
    gold/                  # metrics_basic.py  mrr.py  cac.py  ltv.py
    common/                # paths.py  logger.py  db.py
    main.py                # pipeline orchestrator
  logs/                    # run logs
  .env                     # DATA_DIR / LAKEHOUSE_DIR overrides
  requirements.txt
```

9.1 How to Run

Setup:

```
pip install -r requirements.txt
```

Run full pipeline:

```
python -m src.main
```

Run individual stages:

```
python -m src.bronze.events
python -m src.silver.events
python -m src.gold.metrics_basic
```

Query gold tables via DuckDB:

```
duckdb lakehouse/gold/analytics.duckdb
  SELECT * FROM daily_active_users ORDER BY event_date;
  SELECT * FROM ltv_cac_ratio;
```

10. Trade-offs & Future Work

Area	Current Approach	Production Extension
Incrementality	Full refresh every run	Partition by date + checkpoint file; merge into existing partitions
Format	Plain Parquet	Apache Iceberg for time-travel, schema evolution, and compaction
Orchestration	Sequential Python function calls	Dagster or Airflow for dependency graphs, retries, and alerting
Scale	Pandas in-process	Polars or Spark for data volumes that exceed single-machine memory
Testing	No automated tests	pytest unit tests for each transform function; Great Expectations for data quality gates
MRR precision	price summed per day	Normalise monthly price to daily (price / days_in_month) for true daily MRR