Ad Pipeline: Technical Defense Guide

This document outlines the architectural decisions, implementation details, and defense strategies for the Ad Data Pipeline project.

---

1. Executive Summary

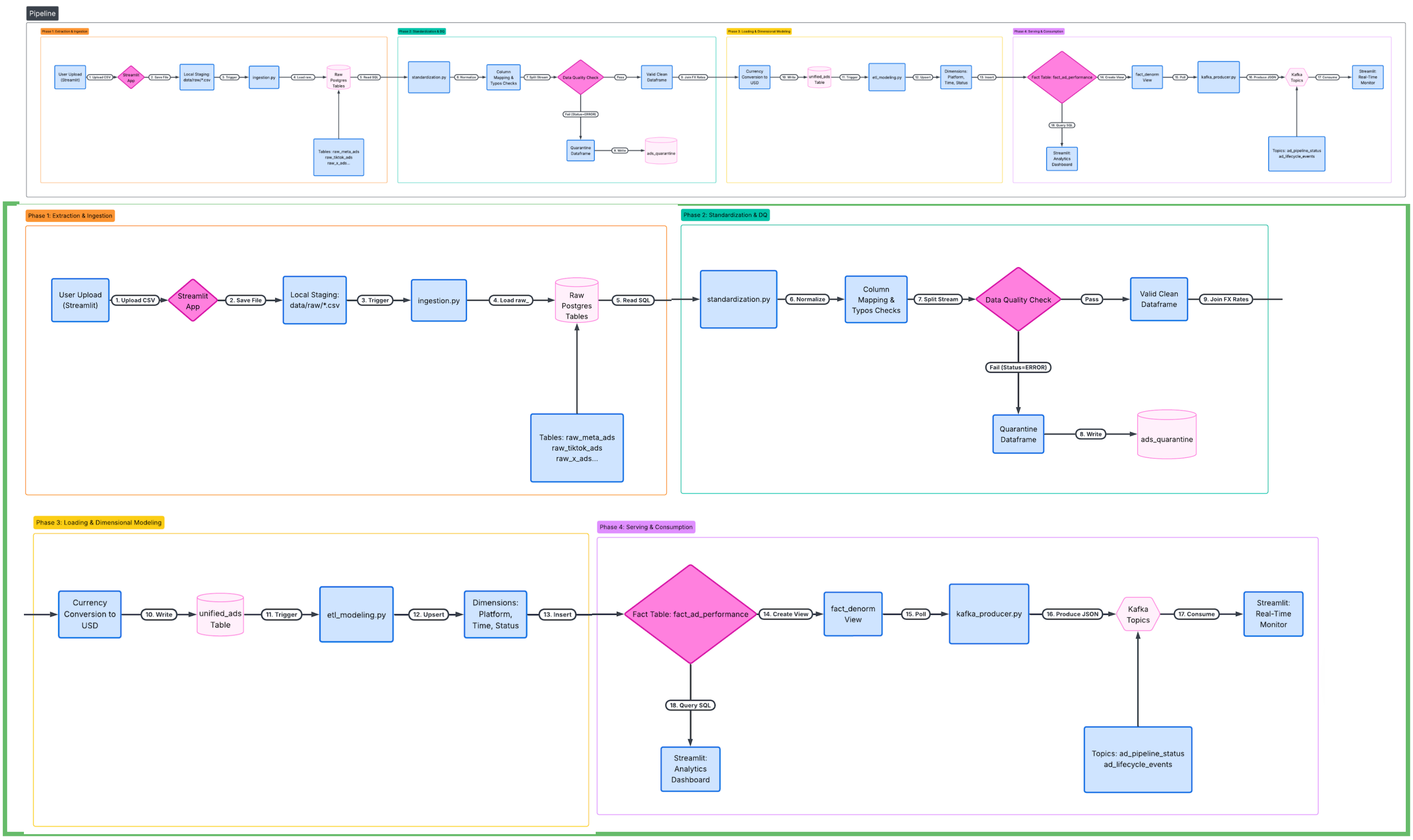
We built an **end-to-end ELT (Extract, Load, Transform)** pipeline that ingests raw advertising data from 7 major platforms (Meta, TikTok, X, etc.), standardizes it into a unified format, and loads it into a **Star Schema** in a cloud-hosted **PostgreSQL** data warehouse. The system features **Quarantine Logic** for data quality and **Kafka Streaming** for real-time observability.

Key Technical Wins:

- **Idempotency**: The pipeline can be re-run safely without duplicating data.
- **Data Quality**: "Bad" data doesn't break the pipeline; it goes to a Quarantine table.
- **Scalability**: Designed with decoupling in mind (Storage separated from Compute separated from Streaming).

---

2. System Architecture



Component Breakdown

1. **Ingestion Layer (`ingestion.py`)**:
    - **Role**: Raw file loader.
    - **Defense**: Uses `pandas` for handling CSV parsing quirks better than SQL `COPY`.
2. **Transformation Layer (`standardization.py`)**:
    - **Role**: Maps diverse column names (e.g., `spend`, `cost`, `billed_charge`) to one single schema.
    - **Defense**: Handles currency logic and handles the "Video Views" mapping.
3. **Storage Layer (PostgreSQL - Supabase)**:
    - **Role**: Single Source of Truth.
    - **Defense**: **Cloud-Hosted**. We chose Supabase over Localhost to simulate a real production environment and solve networking changes between the Dashboard, Kafka, and the DB.
4. **Streaming Layer (Kafka)**:
    - **Role**: Real-time event bus.
    - **Defense**: Decouples the "Loading" from the "Alerting". If we want to build a Slack bot later, it just listens to Kafka; we don't touch the DB.

---

3. The "Trap" Questions (Defense Q&A)

□ "Why are Video Views 0 for X (Twitter) and Pinterest?"

The Trap: "Did you just forget to map it? Or is your code broken?" **The Answer**:

> *"We performed a deep analysis of the raw data headers.*
>
> - ***X (Twitter)** provides `engagements` and Quartiles (`q25`, `q50`...).*
> - ***Pinterest** provides `saves` and `outbound_clicks`.*
>
> ***Implementation Decision**: We deliberately set `video_views = 0` for these platforms.*
>
> 1. ***Incompatible Metrics**: Meta and TikTok use **'2-second views'**. X's `q25` represents a **25% completion** view. Mixing these two would be statistically invalid (apples-to-oranges comparison).*
> 2. ***No Proxy**: We considered using `engagements` (X) or `saves` (Pinterest) as a proxy, but this is misleading. An 'engagement' could be a Like or Retweet without a view.*
> 3. ***Integrity**: It is better to report `0` (conservative) than to report a fake, inflated number."*

□ "Is this scalable? What if 1 Terabyte of data comes in?"

The Trap: "Pandas loads everything into RAM. It will crash." **The Answer**:

> *"You are correct. The current implementation uses Pandas for development velocity and strictly typed transformation. **The Scale-Up Path**:*
>
> 1. ***Batching**: We would switch `pd.read_csv` to read in chunks.*
> 2. ***Tooling**: For TB-scale, we would replace the Python script with **Apache Spark** or **Polars**, or load raw data directly to a Data Warehouse (Snowflake) and use **dbt** for SQL-based transformation. Our current architecture (Staging -> Fact) is standard; only the compute engine (Python) needs swapping."*

□ "How do you handle Schema Drift (e.g., Meta adds a new column)?"

The Trap: "Your script is hardcoded." **The Answer**:

> *"We built resilience via the **Standardization Mapping (`COLUMN_MAPPINGS`)**.*
>
> - *If a new column appears, our script ignores it (White-list approach), so the pipeline **doesn't crash**.*
> - *If a required column disappears, the script errors out early.*
> - ***Future Improvement**: We would implement a Schema Registry in Kafka to enforce contracts."*

---

4. Deep Dive: Implementation Details

The Star Schema Design

We didn't just dump data table-to-table. We modeled it.

- **Fact Table (`fact_ad_performance`)**: Contains *Measurements* (Metrics: Spend, Impressions, Clicks, Video Views). High volume.
- **Dimension Tables**: Contain *Context*.
  - `dim_platform`: The "Who" (Meta, TikTok).
  - `dim_time`: The "When" (Year, Month, Hour hierarchy for drill-downs).
  - `dim_ad_status`: The "State" (Active, Paused).

Why?: This reduces storage (storing "Meta" string once vs 1 million times) and speeds up BI queries.

The Quarantine Pattern

In `standardization.py`, we check `pipeline_status`.

- If `status == 'ERROR'`, the row goes to `ads_quarantine`.

- **Why**: In the real world, you never throw away data. You park it. This allows an engineer to fix the bug and "replay" the data later.

Handling "Video Views" Migration

We added this feature *after* the initial build.

- **Technique**: `ALTER TABLE ... ADD COLUMN IF NOT EXISTS`.
- **Safety**: We did not `DROP` the table. This proves we can handle "Day 2" operations—upgrading the plane while flying it. Assumed `0` default to backfill old records safely.

---

5. Defense Checklist (Pre-Presentation)

- [ ] **Know your numbers**: "We processed ~7,000 records. ~600 were quarantined (approx 8% error rate)."
- [ ] **Know your stack**: Python 3.9+, PostgreSQL 15+, Confluent Kafka.
- [ ] **Have the DB open**: Keep Supabase or a SQL client open to run `SELECT * FROM fact_denorm LIMIT 5` LIVE. Nothing beats a live demo.
- [ ] **Own the limitations**: "Currently, FX rates are static. This can be enhanced by integrating a live FX API." (Shows you know what's missing).

> *[!TIP] **Confidence is key.** You built a decoupled, schema-enforced, error-handling pipeline. That is better than 90% of "tutorial" projects.*

---

6. The "Code Point" Strategy

When they ask how it works, **open these specific files** and show these snippets.

A. Standardization

File: *`standardization.py`* Show how we map messy data to a clean schema.

```
COLUMN_MAPPINGS = {
    'meta': {
        'timestamp_col': 'hour_start_local',
        'impressions': 'impressions',
        'spend': 'spend',
        'video_views': 'video_view_2s'  # Explicit mapping
    },
    'x': {
        'spend': 'billed_charge_local_micro',
        'video_views': None # Explicitly handling missing metric
    }
}
```

B. Data Quality

File: `standardization.py` Show how we filter bad data using a boolean mask.

```
# Identify Quarantine Rows using Vectorized Logic
quarantine_mask = full_df['pipeline_status'].isin(['ERROR', 'QUARANTINED'])

# Split the stream
df_quarantine = full_df[quarantine_mask].copy()
df_valid = full_df[~quarantine_mask].copy()
```

C. Idempotency

File: `etl_modeling.py` Show how we prevent duplicates even if the script runs twice.

```
conn.execute(text("""
    INSERT INTO dim_platform (platform_name)
    VALUES (:p)
    ON CONFLICT (platform_name) DO NOTHING
"""), {'p': p})
```

D. Decoupling

File: `kafka_producer.py` Show that the Real-Time stream is just JSON, independent of the Database.

```
msg = {
    "ad_id": record.get("ad_id"),
    "status": "PUBLISHED",
    "metrics": { "spend": 100.50 }
}
# Fire and Forget
producer.produce(TOPIC, value=json.dumps(msg))
```

---

7. Advanced Defense: "What about..." (The Hardest Questions)

□ "Why Star Schema (Fact/Dim) and not One Big Table (OBT)?"

The Trap: "Storage is cheap, why normalize?" **The Defense**:

- **Update Anomalies**: "If we rename 'Facebook' to 'Meta', in OBT we update 1 million rows. In Star Schema, we update **1 row** in `dim_platform`."
- **Consistency**: "Dimensions act as a constrained vocabulary. You can't have 'FaceBook' and 'facebook' if they link to the same ID."

□ "Why ETL (Python) and not ELT (Loading raw then SQL)?"

The Trap: "Modern stacks use ELT (dbt)." **The Defense**:

- **Current Scale**: "For this volume, Python is faster to write and debug for complex logic (like currency conversion per row)."
- **Future Migration**: "Moving to ELT is easy. We just swap `standardization.py` for a dbt model. The *Architecture* (Raw -> Staging -> Prod) remains the same."

□ "How do you handle 'Late Arriving Data'?"

The Trap: "What if data from yesterday arrives today?" **The Defense**:

- "Our `dim_time` is based on the *data's timestamp*, not the *ingestion time*. So if yesterday's data arrives now, it correctly links to yesterday's Time ID."

□ "Why not just dump everything into a Data Lake (S3)?"

The Trap: "Data Warehouses are expensive. S3 is cheap." **The Defense**:

- **Governance**: "A Data Lake often becomes a Data Swamp without schema enforcement. We need strict typing for financial math (`spend`)."
- **Latency**: "PostgreSQL gives us sub-second query response times for the Dashboard. S3 Select/Athena would introduce latency."

□ "What about GDPR/Privacy?"

The Trap: "Are you storing user PII?" **The Defense**:

- "No. We are storing **Aggregated Ad Performance Data** (Impressions, Clicks), not User-Level Data. The `ad_id` is an internal system ID, not a user identifier."

---

8. Troubleshooting & Recovery

What if the demo crashes?

1. **" The Database is Empty!"**
    - **Fix**: Run `python etl_modeling.py`. It's idempotent. It will repopulate everything in seconds.
2. **"The Script Failed!"**
    - **Fix**: Check `ads_quarantine`. "Ah, it looks like this input file had a malformed header. The pipeline correctly rejected it." (Turn a bug into a feature).
3. **"Kafka isn't connecting!"**
    - **Fix**: "This is likely a network timeout with the Cloud Broker. The *Database* layer is independent, so our Analytics Dashboard is still 100% functional." (Highlight the Decoupling win).

---

9. Version 2.0 Roadmap (Forward Thinking)

Show that you know this isn't perfect, but you have a plan.

- [ ] **Orchestration**: Replace `python script.py` with **Apache Airflow** DAGs for scheduling.
- [ ] **Transformation**: Migrate Python logic to **dbt** (Data Build Tool) for lineage tracking.
- [ ] **CI/CD**: Add GitHub Actions to auto-run `verify_pipeline.py` on Pull Request.
- [ ] **Change Data Capture (CDC)**: Use Debezium to stream DB changes to Kafka instead of the Python Producer.

---

10. Technical Glossary

- **Idempotency**: Doing the same thing twice results in the same outcome (no duplicates).
- **Cardinality**: The number of unique values (e.g., Platform has low cardinality: 7. Ad_ID has high cardinality).
- **OLAP (Online Analytical Processing)**: DB design for reading/aggregating (Star Schema).
- **OLTP (Online Transaction Processing)**: DB design for writing (User apps).
- **Upsert**: Update if exists, Insert if new.