ETL Architecture Deep Dive: Bronze Layer Ingestion Pipeline

1. Executive Summary & Design Philosophy

This document details the architecture and engineering decisions for our Bronze Layer ETL pipeline. The goal was not just to load CSV files, but to build a professional, production-grade system that is:

1. **Resilient:** It must not fail due to "dirty" source data.
2. **Maintainable:** All logic must be easy to read, debug, and modify (i.e., it must be code in Version Control).
3. **Observable:** We must be able to log every run, capture all errors, and identify performance bottlenecks.
4. **Secure:** Credentials must be externalized, and we must avoid insecure server configurations.

The final architecture is a **Python-Orchestrated** model. We deliberately moved all orchestration, scheduling, and error-handling logic *out* of the database and into a Python script. This is a modern, flexible, and scalable design that solves several critical problems we encountered.

2. Architectural Components & File Structure

Our final system is composed of two phases: **One-Time Setup** (run by a DBA) and **Daily Orchestration** (run by an automated process).

Phase 1: One-Time Setup Files

These files are run once to "build the warehouse."

- `00_create_warehouse_schema.sql` : (Admin) Creates the `dw_bronze` , `dw_silver` , and `dw_gold` databases.
- `00a_create_logging_utility.sql` : (Admin) Creates the `etl_log` table for persistent, queryable logging.
- `01_create_bronze_tables.sql` : (Admin/Engineer) Creates the DDL for all our Bronze tables.

Phase 2: Daily Orchestration Files

These files *are* the daily ETL job.

- `requirements.txt` : Defines the Python libraries needed (the "tools").
- `.env` : (Secret) Stores our database credentials and file paths.
- `run_bronze_load.py` : (The "Brain") The Python script that orchestrates the entire daily load.

3. Phase 1: The "One-Time Setup" - Key Decisions

Decision: Why separate schema and table creation (`00_` vs. `01_`)?

Best Practice: Separation of Concerns & Security.

- `00_create_warehouse_schema.sql` is a high-privilege **DBA task**. It creates the "buildings." A daily ETL user should *never* have permission to `DROP DATABASE` .

- `01_create_bronze_tables.sql` is a lower-privilege **Engineer task**. It creates the "shelves" inside the building.
- By splitting them, we enforce security and make the pipeline modular.

Decision: Why create a dedicated `etl_log` table (`00a_...`)?

Best Practice: Observability.

- **Option 1 (Rejected):** Use `SELECT 'message'` or text file logs. This is fine for manual debugging, but the logs are "dumb." You cannot query them to find trends (e.g., "Show me all failed runs in the last 30 days" or "What is the average load time for `crm_sales_details` ?").
- **Option 2 (Chosen):** Create a structured `etl_log` table. This turns our logs into **data**. We can now query our logs just like any other table, build dashboards on them, and precisely monitor for performance bottlenecks and error rates. The `DATETIME(6)` (microsecond precision) on `start_time` / `end_time` is specifically for accurate bottleneck analysis.

Decision: Why are all columns in `01_create_bronze_tables.sql` `VARCHAR(255)` ?

Best Practice: Bronze Layer Resilience.

- **This is the most important principle of the Bronze layer.** Its job is to be a resilient, 1-to-1 copy of the source data, "warts and all."
- **Option 1 (Rejected):** Strict Typing (e.g., `prd_cost DECIMAL(10,2)` , `sls_order_dt DATE`).
  - **Problem:** This is brittle. As we saw in our initial exploration, our source files contain bad data (`''` in a `DECIMAL` column, `'SO43697'` in an `INT` column).
  - **Result:** A strictly-typed `LOAD DATA` command would **fail the entire ETL job** the first time it sees a single piece of "dirty" data.
- **Option 2 (Chosen):** `VARCHAR(255)` for all columns.
  - **Benefit:** The load *will never fail* due to a data type mismatch. It captures `'N/A'` , `''` , or `'SO43697'` as a simple string.
  - **Philosophy:** We are deliberately deferring the problem. **Capturing** data (Bronze layer) and **Cleaning/Typing** data (Silver layer) are two separate jobs. This design ensures the Bronze load always succeeds.

4. Phase 2: The "Daily Orchestrator" - Key Decisions

This is where we made the most critical architectural choice: **moving the orchestration logic from SQL to Python.**

Decision: Why use Python (`run_bronze_load.py`) instead of a SQL script or Stored Procedure?

This was the pivotal decision that solved all our major problems.

- **Option 1 (Rejected):** A Stored Procedure (`sp_run_bronze_load`).
  - **The "Blocker":** Our server (like most modern MySQL instances) has `secure_file_priv = NULL` . This is a security feature that **disables** server-side `LOAD DATA INFILE` . A stored procedure runs *on the server*, so it would be blocked by this rule.

    - **The "Bad Fix":** The only way to make it work would be to (as a DBA) edit the server's `my.cnf` file, set a path, restart the server, and grant `FILE` privileges. This is a high-friction, insecure, and non-portable solution.
    - **Other Cons:** Logic is hidden in the database (bad for Git/version control) and it's hard to do complex logging or connect to other sources (like an API).
- **Option 2 (Chosen):** A Python Orchestration Script.
    - **The "Magic Key":** The Python script is a **client**. By connecting with `allow_local_infile=True`, we can use `LOAD DATA **LOCAL** INFILE`. This client-side command *bypasses* the `secure_file_priv` restriction entirely. The script reads the local file and streams the data to the server.
    - **Superior Error Handling:** Python's `try...except...finally` is far more powerful than SQL's `DECLARE HANDLER`. It can catch *any* error (e.g., "database connection failed," "file not found," "invalid password," *and* SQL errors), not just `SQLEXCEPTION`.
    - **Superior Logging:** Python's `logging` module is the industry standard. It easily supports our "dual logging" (to both the console and a text file) for debugging.
    - **Future-Proof:** If we need to load from a new source next week (e.g., a website API or an S3 bucket), we can just `import requests` or `import boto3` in this script. The architecture is now scalable.

Decision: Why use `.env` (`read_config`)?

Best Practice: Security & Portability.

- **Rejected:** Hard-coding credentials (`password='root@1234'`) in the script. This is a massive security risk. If you commit the code to Git, you have leaked your password.
- **Chosen:** `load_dotenv()` reads from a `.env` file. This file is listed in `.gitignore` (it *never* gets committed). It allows you to have different `.env` files for development, testing, and production. The `sys.exit(1)` call is a "fail-fast" best practice: if a password variable is missing, the script stops immediately.

Decision: Why use `TRUNCATE TABLE` instead of `DROP/CREATE`?

Best Practice: Performance & Permission-Handling.

- Our old SQL script (`01_load_bronze_layer.sql`) used `DROP/CREATE`. This is fine for *development* but bad for *production*.
- **Why `DROP/CREATE` is bad (for daily loads):**
    1. **Slow:** It has to de-allocate all the storage, drop indexes, then re-create the table, re-build indexes, and re-check permissions.
    2. **Destroys Permissions:** If a DBA had set specific `GRANT` permissions on that table, `DROP` would wipe them out.
- **Why `TRUNCATE TABLE` is better (Chosen):**
    1. **Fast:** It's a metadata-only operation. It just moves the table's "high-water mark" back to zero. The table, its indexes, and its permissions all remain, ready to be filled.

2. **Correct for Full Refreshes:** It's the most efficient way to wipe a table completely before a full reload.

Decision: Why `connection.commit()` inside the `for` loop?

Best Practice: Granular Control (A Deliberate Design Choice).

- **Option 1:** Wrap the *entire* `for` loop in one `START TRANSACTION ... COMMIT` (like our old SQL script did). This is an "All-or-Nothing" atomic model. If table 5 fails, tables 1-4 are rolled back. This is very safe.
- **Option 2 (Chosen):** Commit after *each* table. This is a "Partial Success" model. If table 5 fails, tables 1-4 are successfully loaded and committed. For a long-running ETL (e.g., hours), this is often preferred so you don't lose hours of work from a single late-stage failure. Our script is fast, but this demonstrates the granular control Python gives us.

Decision: Why two types of logging (`etl_log` table and `bronze_load.log` file)?

Best Practice: Dual-Mode Observability.

- The `etl_log` **database table** (written by `log_etl_start` / `log_etl_end`) is for **Monitoring & Analysis**. It answers the question, "What was the performance of all runs this week?"