Sales Data Pipeline

This project demonstrates an end-to-end data pipeline using Python, PostgreSQL, and dbt, orchestrated with Docker Compose. It ingests sales data, validates it, loads it into Postgres, and transforms it into analytics-ready models.

Architecture

Services

- Postgres – Warehouse database, initialized with schemas: `raw`, `staging`, `intermediate`, `marts`
- Python ETL – Loads data into `raw.orders` (reads CSV or generates synthetic sales data)
- dbt – Transforms raw data into analytics-ready tables, runs tests, generates lineage/docs

Data Flow

1. Extract – Read CSV or generate synthetic sales orders
2. Validate – Enforce schema, datatypes, business rules, dedupe
3. Load – Bulk load into `raw.orders` in Postgres
4. Transform (dbt):
    - Staging: Type casting, standardization
    - Intermediate: Enrich data (e.g., `line_amount`)
    - Marts: Aggregated facts (`fct_daily_sales`)
5. Test & Document – dbt validates constraints and generates documentation

Approach

The solution is built to simulate a production-grade data platform in a containerized environment. The architecture has three major services orchestrated by Docker Compose: Postgres, Python ETL, and dbt. Each service plays a distinct role in the data lifecycle, from ingestion to transformation to analytics.

The pipeline begins with Postgres, which acts as the central data warehouse. Multiple schemas are pre-created in Postgres: raw for ingestion, staging for typed views, intermediate for enriched transformations, and marts for analytics-ready data. This separation of layers ensures that raw

data is never overwritten and that every downstream layer depends on a progressively cleaner and more meaningful dataset.

The Python ETL service is responsible for data extraction and loading into Postgres. It reads from a CSV if present, or generates synthetic sales data with randomized orders, customers, products, and revenue. Data validation is performed using Pandas for structural checks and Pydantic for schema enforcement. Invalid rows are filtered out, dates and numeric fields are coerced into proper types, and duplicates are removed. Once validated, the data is bulk-loaded into the raw.orders table using efficient Postgres COPY commands. This ensures fast ingestion of thousands of rows.

The dbt service then transforms the raw data into business-ready models. The staging layer creates views that strictly type and cast each column. The intermediate layer enriches the staging data, for example, by computing line_amount as quantity multiplied by unit_price. The marts layer contains the final business models such as fct_daily_sales, which aggregates metrics at the daily level, producing orders, units sold, and gross revenue per day. dbt also handles incremental logic so that only new data is processed on subsequent runs. Alongside the models, dbt runs schema and data quality tests, and generates documentation with data lineage and model catalog.

Orchestration is handled by a PowerShell and Bash script, which ensures deterministic pipeline execution. The script builds Docker images, spins up Postgres, waits for its health check to pass, then runs the Python ETL, followed by dbt models and tests, and finally generates documentation and sample queries. This sequence guarantees that dependencies are respected: Postgres is ready before ETL begins, and transformations only run once raw data is available.

Overall, this architecture mimics a modern data stack: ingestion through an ETL pipeline, storage in a relational warehouse, transformation with dbt, and orchestration to tie the components together. The modularity ensures reproducibility, debuggability, and clear separation of concerns.

Basic Tests for Pipeline Components

The pipeline includes multiple layers of testing and validation to ensure correctness and reliability of results.

Python ETL validations:

- Checks for expected columns and raises errors if required fields are missing.

- Filters only allowed status values such as completed, cancelled, and pending.
- Ensures quantity and unit_price are greater than zero, preventing nonsensical records.
- Deduplicates rows by order identifiers and timestamps, keeping only the most recent version of each order line.
- Converts order_ts and updated_at into proper datetime objects.
- Validates a sample of rows against a Pydantic model called OrderLine, which enforces schema and type correctness. If validation errors occur, they are logged and execution stops.

Database-level checks:

- The ETL ensures that schemas and tables exist by creating them if absent. This makes the pipeline idempotent.
- Data is loaded into a temporary table before being merged into raw.orders. This reduces the risk of partially loaded or inconsistent data if a failure occurs mid-load.
- Indexes are created on updated_at to optimize queries and incremental loading.

dbt tests:

- Staging models test that fields such as order_id, order_line_id, and order_ts are not null.
- Intermediate models check for non-null line_amount values.
- Mart models include uniqueness and not-null tests on keys like sales_date in fct_daily_sales.
- dim_customer tests ensure every customer_id is not null and properly tracked with valid_from and valid_to dates.

Execution proofs:
The logs provide evidence of testing at every stage. For example, ETL successfully generated and validated 5,000 rows:

```
2025-08-22 14:10:20.059 | INFO | __main__:main:21 - Source rows: 5,050
2025-08-22 14:10:20.115 | INFO | __main__:main:23 - Validated rows:
5,000
```

```
2025-08-22 14:10:20.401 | INFO | etl.load:copy_dataframe:55 - Loaded
5,000 rows into raw.orders
2025-08-22 14:10:20.402 | INFO | __main__:main:26 - ETL completed
successfully
```

dbt then ran all models and tests across staging, intermediate, and marts schemas. The logs confirm that every model was created and every test passed:

```
14:10:54 Finished running 1 table model, 1 incremental model in 0
hours 0 minutes and 0.78 seconds
14:10:54 Completed successfully
14:10:59 Finished running 10 tests in 0 hours 0 minutes and 0.75
seconds
14:10:59 Completed successfully
14:10:59 Done. PASS=10 WARN=0 ERROR=0 SKIP=0 TOTAL=10
```

Conclusion:

sample queries demonstrate the correctness of the marts outputs, such as aggregated daily orders, units sold, and gross revenue:

```
sales_date | orders | units_sold | gross_revenue
2025-08-22 |     40 |        157 |       5998.21
2025-08-21 |     72 |        288 |      12250.57
2025-08-20 |     79 |        293 |      11729.76
```