Data Engineering Labs

Final Report

Adam KHALI Ilyas DAHAOUI

February 2026

Contents

1 Introduction

This report documents the work completed in Lab 1 and Lab 2 of the Data Engineering course. Lab 1 built a Python-only ETL pipeline to ingest, clean, and aggregate Google Play Store data. Lab 2 refactored the transformation layer using dbt and DuckDB in order to improve modularity, testability, and analytical modelling practices, while retaining Python for data extraction.

The formal objectives of Lab 2 were:

- Install and configure the development environment, reusing the virtual environment created for Lab 1.
- Install DuckDB, `dbt-core`, and the `dbt-duckdb` adapter, and verify the installation (`dbt -version` should list DuckDB under installed plugins).
- Explore analytical data modelling with emphasis on dimensional modelling and the star schema.
- Structure a transformation pipeline using dbt models and a layered architecture (staging and marts).
- Apply data quality tests using dbt and complement them with Python-based validation.
- Prepare a stable serving layer backed by DuckDB for downstream BI tools.

2 Lab 1: Python Data Pipeline

2.1 Architecture

The Lab 1 pipeline was implemented entirely in Python and organised in three sequential stages.

Stage 1 — Data Ingestion. Raw JSON files produced by the Google Play scraping tool are loaded into memory. The ingestion layer was extended to automatically scan the raw directory for any CSV files dropped alongside the JSON exports, loading them generically to support schema evolution.

Stage 2 — Data Transformation. Application metadata and reviews are cleaned and standardised: column names are normalised, types are cast, and aggregated review metrics are joined with application attributes.

Stage 3 — Data Loading. Processed data is written to JSON files in the `DATA/processed/` folder, producing `apps_metadata_clean.json`, `apps_reviews_clean.json`, `apps_with_metrics.json` and the dimensional outputs (`dim_apps.json`, `dim_categories.json`, `dim_developers.json`, `dim_date.json`, `fact_reviews.json`) later reused by dbt in Lab 2.

Figure 1 shows the pipeline execution log, confirming all three stages completed with 26 app records and 96 862 review records processed.

```
PS C:\Users\ka903\Desktop\Data_Engineering-main\dbt_project> cd C:\Users\ka903\Desktop\Data_Engineering-main\src
>> python pipeline.py
============================================================
STARTING DATA PIPELINE
============================================================
Started at: 2026-02-21 16:00:18


============================================================
STAGE 1: DATA INGESTION
============================================================
Ingesting apps metadata...
Loaded 26 app records
Ingesting apps reviews...
Loaded 96863 review records

============================================================
STAGE 2: DATA TRANSFORMATION
============================================================
Cleaning apps metadata...
Cleaned 26 app records
Cleaning apps reviews...
Cleaned 96862 review records

============================================================
DATA QUALITY CHECKS
============================================================
No obvious quality issues detected.
Loading SCD2 history from C:\Users\ka903\Desktop\Data_Engineering-main\DATA\processed\apps_metadata_scd2.json
Loading from C:\Users\ka903\Desktop\Data_Engineering-main\DATA\processed\apps_reviews_clean.json

Aggregating data for analytics using current snapshot...
Transforming data for analytics...
Created 26 analytics-ready records

============================================================
STAGE 3: DATA LOADING
============================================================
Saved 26 records to apps_metadata_clean.json
Saved 26 records to apps_metadata_scd2.json
Saved 96862 records to apps_reviews_clean.json
Saved 26 records to apps_with_metrics.json
Saved 26 records to dim_apps.json
Saved 1 records to dim_categories.json
Saved 23 records to dim_developers.json
Saved 2601 records to dim_date.json
Saved 96862 records to fact_reviews.json
```

Figure 1: Python pipeline execution

In Lab 2, the architecture evolved: Python continues to handle extraction and initial loading, while all transformation and modelling logic is delegated to dbt. DuckDB serves as both the execution engine and the unified analytical storage layer, and can be queried directly by BI tools without additional ETL steps.

2.2 Data Quality Analysis

An exploratory analysis in Lab 1 revealed several issues.

1. Missing or null fields (`developer`, `updated`, etc.).
2. Inconsistent data types: install counts stored as strings, malformed timestamps.
3. Large JSONL review files causing memory pressure when loaded into Python.
4. Duplicate and orphaned reviews were not detected.
5. Lack of structured validation leading to silent failures.

Midway through the lab, four CSV files were dropped into the raw folder to simulate real-world chaos scenarios.

- `note_taking_ai_reviews_batch2.csv` — a second review batch including deliberate duplicate `reviewId` values.
- `note_taking_ai_reviews_dirty.csv` — rows containing the string `"five"` in the score column, negative values, invalid timestamps, and `NULL` markers.
- `note_taking_ai_reviews_schema_drift.csv` — alternate column names such as `appTitle` and `rating` instead of the expected names.
- `note_taking_ai_apps_updated.csv` — updated application metadata with missing values and renamed fields.

These files were used to test deduplication, schema drift handling, and robust type parsing. The ingestion logic was extended to scan the raw directory automatically, read CSV files generically, and normalise column names and values, demonstrating resilience to evolving data sources.

2.3 Pipeline Fragility

The most fragile component was the absence of deduplication logic for reviews. Rerunning the extraction step appended duplicate records, corrupting aggregated metrics. Hard-coded field names also made the pipeline vulnerable to schema changes. These two weaknesses directly motivated the architectural changes introduced in Lab 2.

3 Enhancements Introduced in Lab 2 — Python Layer

3.1 Incremental Loading

A `merge_reviews` function was implemented to merge new review records with existing processed data using `review_id` as a natural key. Records already present in the processed file are skipped; only genuinely new records are appended. This ensures idempotent pipeline execution and eliminates the duplication problem identified in Lab 1.

3.2 Slowly Changing Dimensions — SCD Type 2

Application metadata changes are now tracked using SCD Type 2 logic. When an attribute such as category name or developer changes between runs, the existing record is closed and a new version is inserted. The history table therefore includes the following columns.

- `start_date` — when this version of the record became active.
- `end_date` — when it was superseded (`null` for the current version).
- `current_flag` — a boolean convenience flag for filtering the latest record.

This enables historical analysis of attribute changes over time, such as tracking how an application's category or average rating evolved across successive scraping runs.

3.3 Data Quality and Testing

A dedicated module `quality.py` performs validation checks before the loading stage: it reports missing identifiers, type mismatches, and ratings outside the valid 1–5 range. The pipeline halts on critical failures, turning silent errors into explicit exceptions.

A PyTest suite covers utility functions, ingestion logic, SCD 2 update and deletion scenarios, and schema drift handling. Figure 2 shows all seven tests passing, with four non-blocking deprecation warnings related to the `datetime.utcnow()` API.

```
(.venv) PS C:\Users\ka903\Desktop\Data_Engineering-main> python -m pytest -q
.......                                                                  [100%]
=================================== warnings summary ===================================
tests/test_pipeline.py::test_pipeline_incremental
tests/test_pipeline.py::test_pipeline_incremental
  C:\Users\ka903\Desktop\Data_Engineering-main\src\utils.py:35: DeprecationWarning: datetime.da
tetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezone-aware
 objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
    timestamp = datetime.utcnow().isoformat()

tests/test_utils.py::test_scd2_update_insert_and_change
  c:\Users\ka903\Desktop\Data_Engineering-main\tests\test_utils.py:29: DeprecationWarning: date
time.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezon
e-aware objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
    now = datetime.utcnow().isoformat()

tests/test_utils.py::test_scd2_update_deletion
  c:\Users\ka903\Desktop\Data_Engineering-main\tests\test_utils.py:49: DeprecationWarning: date
time.datetime.utcnow() is deprecated and scheduled for removal in a future version. Use timezon
e-aware objects to represent datetimes in UTC: datetime.datetime.now(datetime.UTC).
    now = datetime.utcnow().isoformat()

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
7 passed, 4 warnings in 0.51s
(.venv) PS C:\Users\ka903\Desktop\Data_Engineering-main>
```

Figure 2: PyTest output — 7 passed, 4 warnings in 0.51 s

4 Data Modelling

Before building the dbt pipeline, the Kimball four-step dimensional modelling methodology was applied to design the analytical star schema.

4.1 Theoretical Background

Dimensional modelling structures data specifically for analytics and decision-making rather than transactional processing. It organises data around business processes (facts) and their descriptive context (dimensions), making analytical queries easier to write and more efficient to execute.

The **star schema** is the most common physical representation. A central fact table stores measurable events and foreign keys to surrounding dimension tables, which provide the descriptive attributes needed for filtering, grouping, and drill-down. When dimension tables themselves contain hierarchies that are normalised into separate tables, the result is a **snowflake schema**.

The **Kimball methodology** advocates four sequential steps: identify the business process, declare the grain, identify the dimensions, and identify the facts. We adopted this approach because our objective is analytical exploration rather than transactional consistency.

4.2 Step 1 — Business Process

The business process under analysis is the **submission of user reviews** for AI note-taking applications on the Google Play Store. This event is measurable, repeatable, and captures both quantitative attributes (star rating, thumbs-up count) and rich descriptive context (which application, which developer, which category, when).

4.3 Step 2 — Grain Declaration

> *One row in the fact table represents one review event submitted by one user for one specific application at a given point in time.*

Each review is uniquely identified by its `review_id`. The grain is at the individual review level — no pre-aggregation is performed in the fact table.

4.4 Step 3 — Dimensions

Table 1 lists the four dimensions identified, along with their business meaning, key attributes, and source dataset.

Table 1: Identified dimensions

Dimension	Business meaning	Key attributes	Source
`dim_apps`	Descriptive information about each application	app_id, title, avg_rating, install count, price, price tier	apps_metadata.json
`dim_categories`	Genre or category grouping (sits above apps in the hierarchy)	category_sk, category_name	apps_metadata.json (genre field)
`dim_developers`	Who built the application	developer_sk, developer_name, primary_category	apps_metadata.json (developer field)
`dim_date`	When the review occurred; enables time-based roll-ups	date_key (YYYYMMDD), year, month, quarter, week, day_name, is_weekend, year_month	Generated from review date range

Dimension hierarchy. `dim_apps` and `dim_categories` form a parent–child hierarchy: each application belongs to exactly one category, and `dim_apps` carries a foreign key to `dim_categories`. This introduces a snowflake sub-structure within an otherwise star schema.

Conformed date dimension. `dim_date` is a conformed dimension — designed to be reusable across any future fact table in the same data warehouse. Following the Kimball convention, its primary key is an integer in `YYYYMMDD` format (e.g. `20260211`) rather than a sequential surrogate. This allows direct human-readable interpretation of the key in the fact table and enables efficient date range filtering without a join.

4.5 Step 4 — Facts and Measures

Table 2 lists the quantitative measures stored in the fact table, together with their additivity classification and valid aggregations.

Table 2: Identified facts and their aggregation semantics

Measure	Description	Additivity	Valid aggregations
`rating`	Star rating (1–5) given by the reviewer	Fully additive	AVG, COUNT, distribution
`thumbs_up_count`	Community upvotes on the review	Fully additive	SUM, AVG
`is_positive_review`	Flag: 1 if rating ≥ 4, else 0	Semi-additive	SUM (count of positive reviews)
`is_negative_review`	Flag: 1 if rating ≤ 2, else 0	Semi-additive	SUM (count of negative reviews)

4.6 Bus Matrix

Table 3 is the Bus Matrix formalising the relationship between the business process and the dimensions by which its facts can be analysed. A tick () indicates that the business process is analysable by the given dimension.

Table 3: Bus Matrix

Business Process	dim_apps	dim_categories	dim_developers	dim_date
User Review Events				

4.7 Star Schema Design

The Bus Matrix yields one fact table (`fact_reviews`) surrounded by four dimension tables. Table 4 summarises the resulting schema. Because `dim_apps` carries a foreign key to `dim_categories`, the overall structure is technically a **snowflake schema**.

Table 4: Star schema — table definitions

Table	Key columns
`fact_reviews`	review_sk (PK); app_sk, developer_sk, category_sk, date_key (FKs); rating, thumbs_up_count, is_positive_review, is_negative_review, reviewed_at
`dim_apps`	app_sk (PK); app_id (natural key); app_title, developer_name, genre, avg_rating, install_count, price, price_tier; category_sk, developer_sk (FKs to parent dimensions)
`dim_categories`	category_sk (PK); category_name
`dim_developers`	developer_sk (PK); developer_name, primary_category
`dim_date`	date_key YYYYMMDD integer (PK); full_date, year, month, quarter, week_of_year, day_name, is_weekend, year_month, year_quarter

Schema validation against analytical needs. All analytical questions defined during the modelling exercise can be expressed as aggregations on `fact_reviews` filtered or grouped by the four dimensions. For example: a monthly average rating trend requires `AVG(rating)` grouped by `dim_date.year_month`; a developer performance comparison requires `AVG(rating)` and `COUNT(*)` grouped by `dim_developers.developer_name`; a category breakdown requires `AVG(rating)` grouped by `dim_categories.category_name`. The declared grain — one row per review — is respected in all joins.

5 dbt & DuckDB Pipeline

5.1 Environment Setup and Project Configuration

The dbt project was initialised with `dbt init`, selecting DuckDB as the database adapter. This created a project folder containing `dbt_project.yml`, sample model files, and a connection profile. The `profiles.yml` file was configured to point to a local DuckDB file at `data/db/playstore.duckdb` with four execution threads.

Materialisation conventions were set in `dbt_project.yml` to enforce a clear separation between layers: staging models are materialised as **views** (lightweight, always current, no storage cost), while mart models are materialised as **tables** (persisted for fast BI queries).

A connection sanity check was performed by creating a minimal model containing `SELECT 1 AS ok, current_timestamp AS created_at` and running it via `dbt run`. The resulting DuckDB table was verified using the DuckDB CLI, confirming that dbt could create objects in the database.

5.2 Staging Layer

Two source models expose the raw JSON files to dbt using DuckDB's `read_json_auto()` function, passing the file paths as dbt variables so the pipeline can be invoked with different data locations without modifying model code.

Two staging models then standardise the raw data. For `stg_apps`, the key transformations are: renaming `appId` to `app_id` and `title` to `app_title`; casting the score to `DOUBLE` and ratings to `BIGINT`; stripping commas and the `+` character from the installs string before casting to `BIGINT`; defaulting null prices to zero; and generating an MD5 surrogate key from the lowercase, trimmed `app_id`. Rows with a null or empty `app_id` or title are filtered out.

For `stg_reviews`, the model uses `COALESCE` to resolve alternate column names produced by the schema drift file (`rating` vs `score`, `review_time` vs `at`). `TRY_CAST` combined with a range filter silently drops the string value `"five"` and the `-1` rating from the dirty data file. Rows with a null `review_id`, null `app_id`, null rating, or unparseable timestamp are excluded. Surrogate keys are generated for both the review and the parent application.

Schema tests are declared in `schema.yml` for both staging models, covering: uniqueness and not-null constraints on `app_id` and `review_id`; a referential integrity test ensuring every review references a known application; and an accepted-values test confirming ratings are within the 1–5 range. Figure 3 shows all five tests passing.

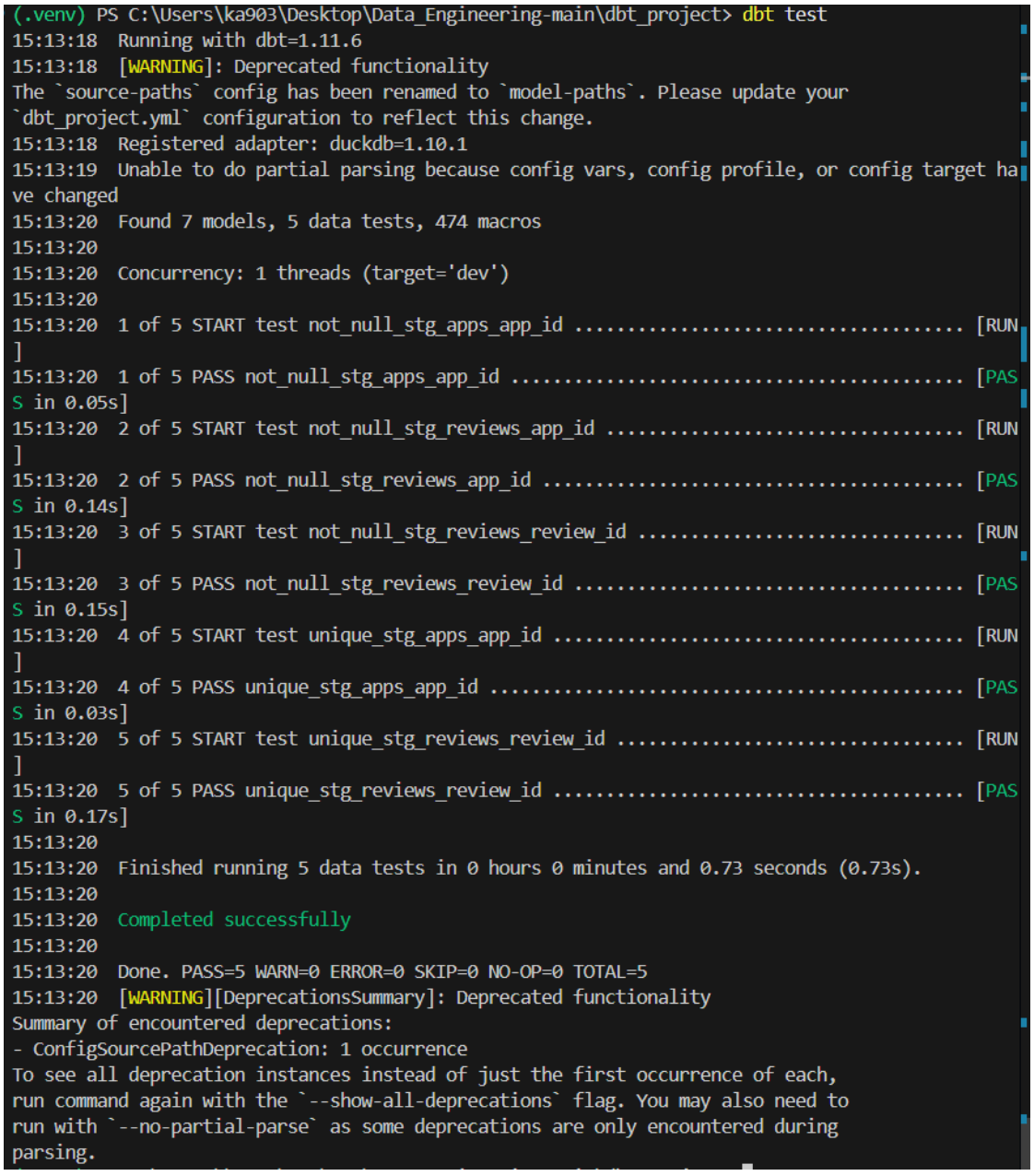

Figure 3: dbt test — PASS=5, WARN=0, ERROR=0, TOTAL=5

5.3 Dimensional Models

Each dimension is built by selecting from the staging views, generating an MD5 surrogate key, and retaining the natural key for downstream joins.

dim_categories selects distinct genre values from `stg_apps`, applying MD5 to the lowercase, trimmed genre string to form the surrogate key.

dim_developers selects distinct developer names together with their primary category, applying the same MD5 key derivation.

dim_apps joins `stg_apps` with both `dim_categories` and `dim_developers` to embed the dimension foreign keys. It also derives a `price_tier` column (*Free* or *Paid*), implementing the apps-to-categories hierarchy that produces the snowflake sub-structure.

dim_date is generated via a recursive CTE that produces one row per calendar day spanning the full range of review timestamps. The primary key follows the Kimball convention: an integer in `YYYYMMDD` format derived with `STRFTIME`. Additional attributes include year, month, quarter, week of year, day name, a weekend flag, and formatted year-month and year-quarter labels for easy BI grouping.

Schema tests for the dimension layer include uniqueness on all surrogate keys and referential integrity tests verifying that every `category_sk` in `dim_apps` resolves to a row in `dim_categories`, and every `developer_sk` resolves to a row in `dim_developers`.

5.4 Fact Table

`fact_reviews` is the analytics-ready fact table at the declared grain. It retrieves data from `stg_reviews` and joins to all four dimension tables using their surrogate keys. The join to `dim_date` is performed by casting the review timestamp to a date and matching it to `dim_date.full_date`. Both joins use `INNER JOIN` to automatically exclude any review that cannot be resolved to a valid dimension record, ensuring referential integrity without a separate filter step. Two derived flag columns, `is_positive_review` and `is_negative_review`, are added as computed measures.

Referential integrity tests are declared in `schema.yml` for all four foreign keys, and an accepted-values test validates that the rating column contains only integers in the range 1 to 5.

Figure 4 shows the complete `dbt run` output: two view models (staging) and five table models (three dimensions, the date dimension, and the fact table) all built successfully in 2.01 seconds.

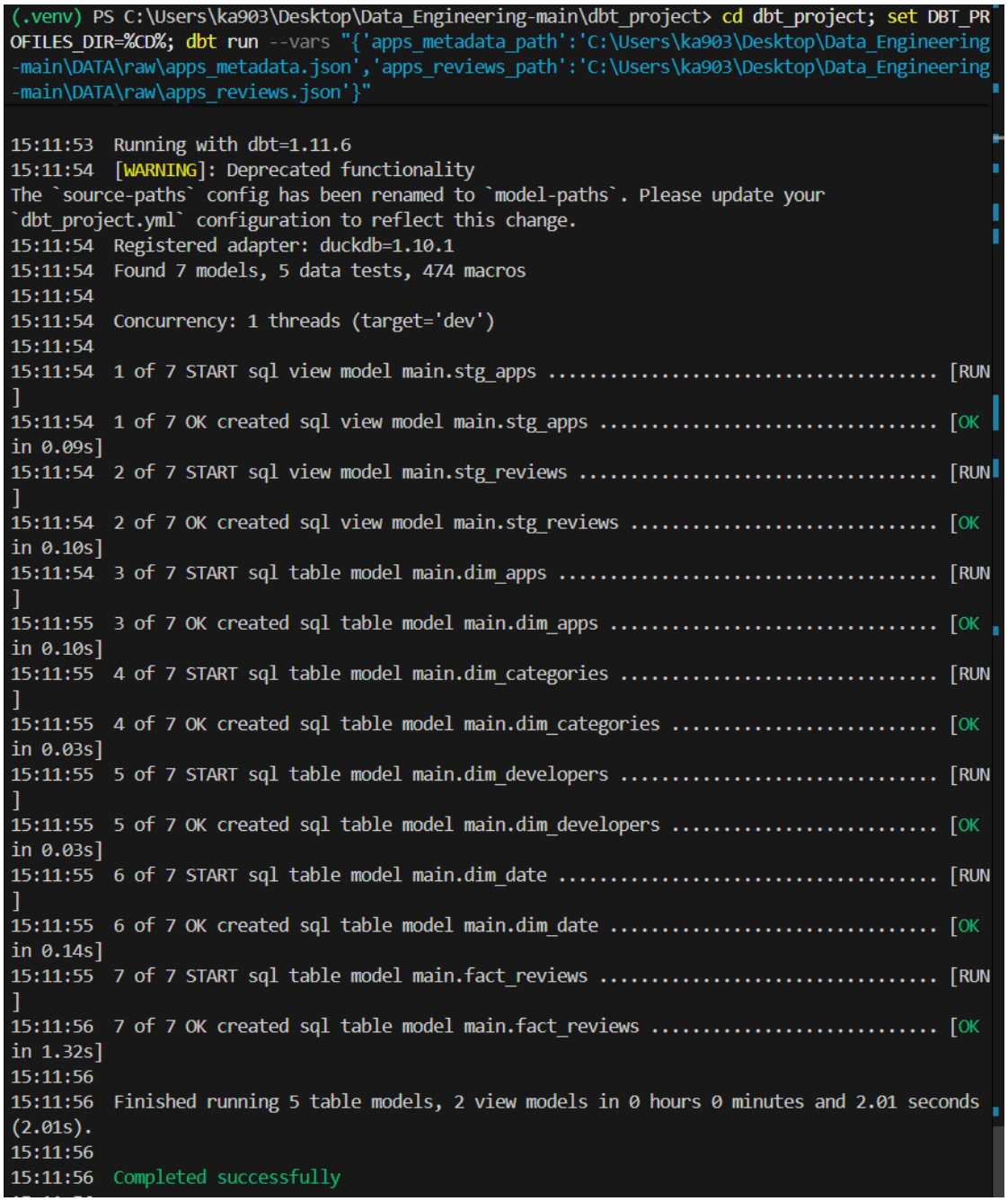

Figure 4: dbt run — 7 models OK (2 views, 5 tables) in 2.01 s

5.5 Serving Layer and BI Consumption

The DuckDB file produced by dbt can be connected directly to BI tools. For Windows users, a DuckDB ODBC driver enables connection from Power BI; for macOS or Linux users, Metabase can be pointed at the `.duckdb` file directly.

Once connected, the BI tool's logical model view was explored to verify that foreign key relationships were detected correctly. Any missing relationships were remapped manually. The star schema enables the following analytical queries without any additional transformation.

- **Monthly average rating trend:** average rating grouped by `dim_date.year_month`,

ordered chronologically.

- **Developer performance comparison:** average rating, total review count, and count of positive reviews grouped by `dim_developers.developer_name`.
- **Category breakdown:** average rating and review count grouped by `dim_categories.category_`

6 Chaos Engineering for Data Pipelines

6.1 Incremental Loading with dbt

The initial pipeline rebuilt `fact_reviews` entirely on every run, which is unacceptable at production scale given the 96 862-review dataset. The fact model was converted to an incremental materialisation using `review_id` as the unique key and the merge incremental strategy.

When the `is_incremental()` condition is active, the model filters the source data to include only reviews whose timestamp is greater than the maximum timestamp already present in the target table. On the first run the full table is built; on subsequent runs only the genuinely new records are processed and merged. Because the unique key prevents duplicates, the model is fully idempotent: running it multiple times with the same source data produces identical results.

The incremental behaviour was tested by appending new records to the reviews JSON file and re-running the pipeline. Only the newly added rows were processed, confirming correct watermark-based filtering.

Comparison with the Python approach. In Lab 1, the `merge_reviews` function achieved a similar outcome but required loading the full processed JSON file into memory to perform the merge in Python. The dbt incremental model delegates this entirely to DuckDB's SQL engine, which scales to arbitrarily large datasets without application-level state management.

6.2 Slowly Changing Dimensions — SCD Type 2 with dbt Snapshots

Until Lab 2, `dim_apps` represented only the current state of each application: every run silently overwrote previous values. To track historical changes — for example, when an application's category or developer name changes — SCD Type 2 was implemented using dbt snapshots.

A snapshot file was created in the `snapshots/` folder using the **check strategy**. The tracked columns are: `app_title`, `developer_name`, `genre`, `avg_rating`, `install_count`, and `price`. On the first run, dbt snapshots the full staging table. On subsequent runs it compares the tracked column values: if any value has changed, the existing row is closed by setting `dbt_valid_to` to the current timestamp, and a new row is inserted with `dbt_valid_to` equal to null.

A downstream dimension model `dim_apps_scd` reads from the snapshot table, preserves the validity timestamps, and adds an `is_current` convenience flag (true where `dbt_valid_to` is null) for simpler querying of the latest version.

To verify the behaviour, the genre of one application was modified in the source JSON file. After re-running `dbt snapshot` and then `dbt run`, the snapshot table contained two rows for that application: one historical version with a closed `dbt_valid_to` timestamp, and one current version with `dbt_valid_to` equal to null.

Linking reviews to historical app versions. Because `fact_reviews` joins to the current `dim_apps`, it can only access the latest version of each application by default. To support true historical analysis — for example, determining which category an application belonged to at the time of each review — `fact_reviews` should instead join to `dim_apps_scd` with a temporal validity predicate. This predicate checks that the review's timestamp falls between the record's `valid_from` and `valid_to` values, using a far-future sentinel date in place of a null `valid_to`.

Comparison with the Python approach. In Lab 1, the `current_flag`, `start_date`, and `end_date` fields were managed by custom Python functions in `utils.py`. While functional, this approach required explicit merge logic for every tracked entity and was difficult to extend. The dbt snapshot declaratively specifies *what* to track and delegates *how* to dbt's built-in merge logic, eliminating the risk of implementation errors.

6.3 Schema Drift and Dirty Data

Table 5 summarises each chaos scenario introduced in the lab and how the dbt architecture addresses it compared to the Lab 1 Python pipeline.

Table 5: Chaos engineering scenarios — Python vs. dbt

Scenario	Lab 1 — Python	Lab 2 — dbt
Schema drift	Required manual column-name mapping in ingestion code; any new variant broke the pipeline	`COALESCE(reviewId, review_id)` in the staging model handles both naming conventions transparently
Dirty data	Custom `try/except` blocks and explicit range checks in `quality.py`	`TRY_CAST` combined with a `WHERE rating BETWEEN 1 AND 5` filter silently drops invalid rows during staging
Duplicate reviews	`merge_reviews` loaded the full processed JSON into memory to deduplicate	Incremental model with `unique_key = 'review_id'` and merge strategy deduplicates at the database engine layer
New review batch	Full pipeline re-run risked appending duplicates if called repeatedly	The `is_incremental()` watermark filter processes only records newer than the existing maximum timestamp

7 Conclusion

These two labs provided hands-on experience designing a complete data engineering pipeline and progressively refactoring it towards modern best practices. The key improvements introduced in Lab 2 are summarised below.

- **Reproducibility.** dbt models are versioned SQL; rebuilding the entire pipeline from scratch requires a single `dbt run` command.
- **Data quality.** Schema tests (not-null, unique, referential integrity, accepted values) catch issues at every layer before they reach BI tools.
- **Modularity.** Each model has a single responsibility; changes propagate downstream only through declared `ref()` dependencies, making the DAG explicit and auditable.
- **Scalability.** Incremental models process only new data on subsequent runs, making the pipeline viable for much larger datasets without full table rebuilds.
- **Historical tracking.** dbt snapshots implement SCD Type 2 with zero custom merge code, replacing the fragile Python SCD logic.
- **Resilience to chaos.** Schema drift and dirty data are handled declaratively in the staging layer rather than defensively in application code, reducing maintenance burden.

The resulting architecture cleanly separates ingestion (Python), transformation (dbt + DuckDB), and serving (DuckDB connected to a BI tool), and is directly extensible to the full Google Play Store dataset.