Data Engineering Labs 1 & 2

Final Report

Adam KHALI   Ilyas DAHAOUI

February 2026

Contents

1 Final Architecture

1.1 Overview

Our final pipeline is a three-layer architecture that cleanly separates ingestion, transformation, and serving. Python handles extraction and initial loading from the Google Play Store API; dbt Core orchestrates all transformation and modelling logic; DuckDB acts as both the execution engine and the unified analytical storage layer.

Layer	**Technology**	**Responsibility**
Ingestion	Python (`pipeline.py`)	API scraping, raw JSON/CSV output
Transformation	dbt Core	Staging, dimensions, fact table, tests
Storage	DuckDB (`playstore.duckdb`)	Unified analytical store and query engine
Serving	DuckDB → BI tool	Direct connection, no additional ETL

The Python pipeline runs first and writes raw outputs to `DATA/raw/`. dbt is then invoked with the paths to those files passed as runtime variables, and builds seven models in dependency order: two staging views, three dimension tables, one conformed date dimension, and the fact table.

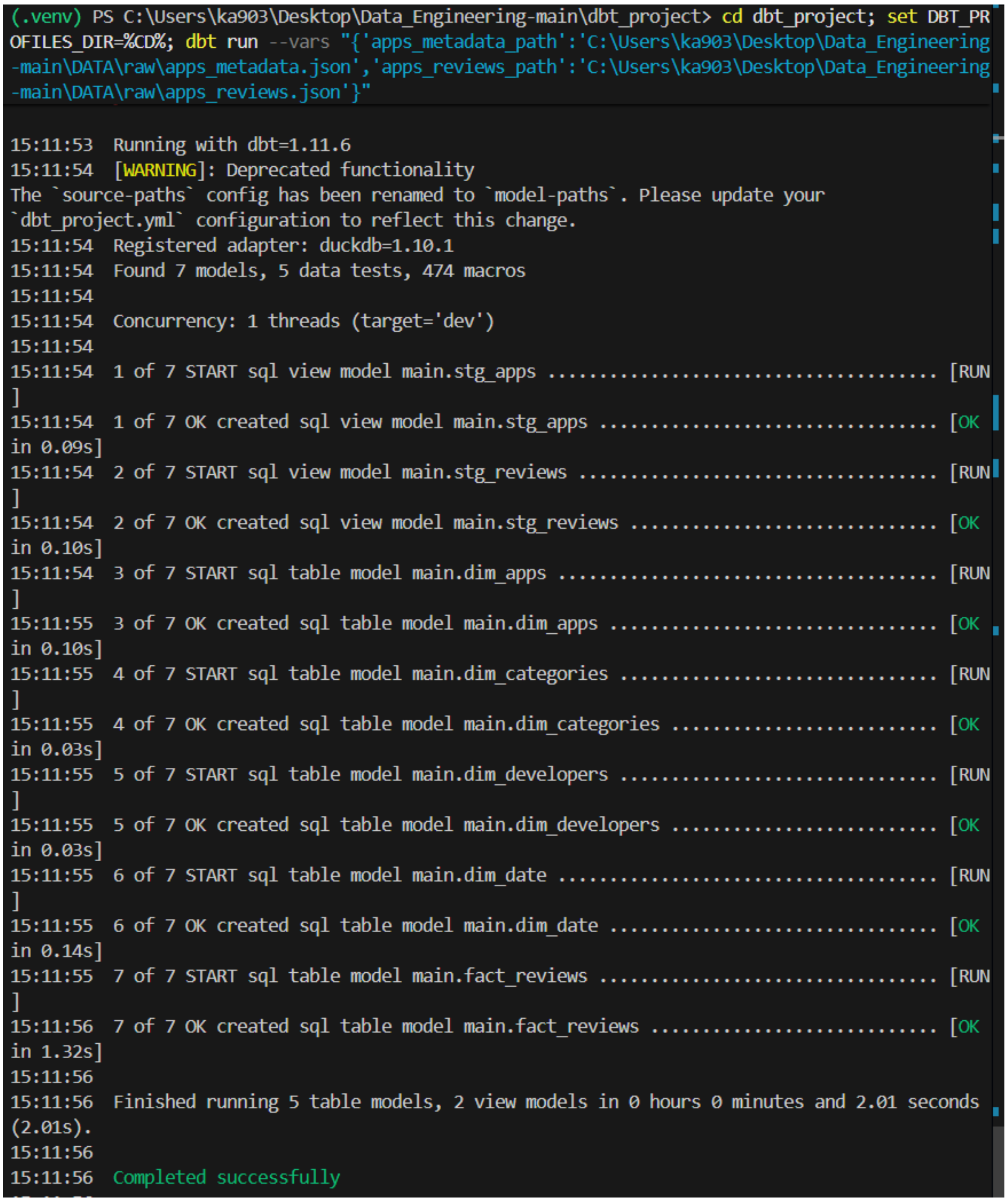
1.2 dbt Model Lineage

The seven models are built in the following order, enforced through `ref()` dependencies declared inside each model:

`src_apps`, `src_reviews` → `stg_apps`, `stg_reviews` → `dim_categories`,
`dim_developers`, `dim_date` → `dim_apps` → `fact_reviews`

Staging models are materialised as **views** — they carry no storage cost and always reflect the current raw data. All mart models are materialised as **tables**, persisted in DuckDB for fast BI queries. This convention is enforced globally in `dbt_project.yml` so that individual models do not need to declare their own materialisation.

Figure 1 shows the successful `dbt run` output confirming all seven models completed without errors.



```
(.venv) PS C:\Users\ka903\Desktop\Data_Engineering-main\dbt_project> cd dbt_project; set DBT_PR
OFILES_DIR=%CD%; dbt run --vars "{'apps_metadata_path':'C:\Users\ka903\Desktop\Data_Engineering
-main\DATA\raw\apps_metadata.json','apps_reviews_path':'C:\Users\ka903\Desktop\Data_Engineering
-main\DATA\raw\apps_reviews.json'}"

15:11:53  Running with dbt=1.11.6
15:11:54  [WARNING]: Deprecated functionality
The `source-paths` config has been renamed to `model-paths`. Please update your
`dbt_project.yml` configuration to reflect this change.
15:11:54  Registered adapter: duckdb=1.10.1
15:11:54  Found 7 models, 5 data tests, 474 macros
15:11:54
15:11:54  Concurrency: 1 threads (target='dev')
15:11:54
15:11:54  1 of 7 START sql view model main.stg_apps ...................................... [RUN
]
15:11:54  1 of 7 OK created sql view model main.stg_apps ................................. [OK
in 0.09s]
15:11:54  2 of 7 START sql view model main.stg_reviews ................................... [RUN
]
15:11:54  2 of 7 OK created sql view model main.stg_reviews .............................. [OK
in 0.10s]
15:11:54  3 of 7 START sql table model main.dim_apps ..................................... [RUN
]
15:11:55  3 of 7 OK created sql table model main.dim_apps ................................ [OK
in 0.10s]
15:11:55  4 of 7 START sql table model main.dim_categories ............................... [RUN
]
15:11:55  4 of 7 OK created sql table model main.dim_categories .......................... [OK
in 0.03s]
15:11:55  5 of 7 START sql table model main.dim_developers ............................... [RUN
]
15:11:55  5 of 7 OK created sql table model main.dim_developers .......................... [OK
in 0.03s]
15:11:55  6 of 7 START sql table model main.dim_date ..................................... [RUN
]
15:11:55  6 of 7 OK created sql table model main.dim_date ................................ [OK
in 0.14s]
15:11:55  7 of 7 START sql table model main.fact_reviews ................................. [RUN
]
15:11:56  7 of 7 OK created sql table model main.fact_reviews ............................ [OK
in 1.32s]
15:11:56
15:11:56  Finished running 5 table models, 2 view models in 0 hours 0 minutes and 2.01 seconds
(2.01s).
15:11:56
15:11:56  Completed successfully
```

Figure 1: `dbt run` output — 7 models built successfully in 2.01 s. Models execute in dependency order enforced by `ref()` declarations: the two staging views complete first (under 0.10 s each), followed by the three dimension tables and the date dimension (0.03–0.14 s), and finally the fact table (1.32 s, reflecting the cost of joining across 96 862 review records). Every model reports `[OK]`, and the run summary confirms `Completed successfully` with no warnings that affect correctness.

1.3 Star Schema Design

The star schema was designed by applying the Kimball four-step methodology before writing any SQL.

Business process: the submission of a user review for an AI note-taking application on the Google Play Store. This is a measurable, repeatable event that produces both quantitative values (the star rating, thumbs-up count) and descriptive context (which app, which developer, which category, when).

Grain declaration: one row in `fact_reviews` represents one review event submitted by one user for one specific application at a specific point in time. Each review is uniquely identified by its `review_id`, and no pre-aggregation occurs in the fact table.

The resulting schema contains one central fact table surrounded by four dimensions:

Table	**Type**	**Key columns and measures**
`fact_reviews`	Fact	`review_sk` (PK); `app_sk`, `developer_sk`, `category_sk`, `date_key` (FKs); rating, thumbs_up_count, is_positive_review, is_negative_review
`dim_apps`	Dimension	`app_sk` (PK, MD5 hash); `app_id` (natural key); title, avg_rating, install_count, price, price_tier, `category_sk`, `developer_sk`
`dim_categories`	Dimension	`category_sk` (PK, MD5); category_name
`dim_developers`	Dimension	`developer_sk` (PK, MD5); developer_name, primary_category
`dim_date`	Conformed dimension	`date_key` in YYYYMMDD integer format (PK); year, month, quarter, week_of_year, day_name, is_weekend, year_month

`dim_apps` carries foreign keys to both `dim_categories` and `dim_developers`, forming a snowflake sub-hierarchy within an otherwise flat star schema. `dim_date` is a conformed dimension — it is designed to be shared across any future fact table in the warehouse. Its primary key follows the Kimball YYYYMMDD integer convention (e.g. `20260211`), which allows the date to be read directly from the fact table without a join, and enables efficient range filtering at the database level.

1.4 Bus Matrix

Business Process	**dim_apps**	**dim_categories**	**dim_developers**	**dim_date**
User Review Events				

The Bus Matrix confirms that all four dimensions are applicable to the single business process, producing one fact table. It also established `dim_date` as a conformed dimension: because it must be reusable across any future fact table, it cannot be a derived column — it must exist as a standalone, pre-populated table.

1.5 Additional Features

Beyond the base lab requirements, we implemented:

- **SCD Type 2 dimension** (`dim_apps_scd`) backed by a dbt snapshot, tracking historical changes to app metadata with `dbt_valid_from`, `dbt_valid_to`, and an `is_current` convenience flag (detailed in Section 2.2).
- **Temporal fact join** — `fact_reviews` can be linked to `dim_apps_scd` using a validity predicate so that each review is analysed against the version of the app that was current at review time, enabling true point-in-time historical analysis.
- **Python quality module** (`quality.py`) — a pre-dbt validation step that halts the pipeline on critical data failures before any record reaches DuckDB.
- **PyTest suite** — seven unit and integration tests covering utility functions, SCD 2 insert, update, and delete scenarios, and incremental pipeline behaviour.

2 Implementation of Key Engineering Concepts

2.1 Incremental Loading

The problem. The original `fact_reviews` model was materialised as a plain table, meaning every `dbt run` rebuilt the entire table from scratch. With 96 862 review records this is computationally wasteful, and at larger scales it becomes prohibitive. A separate but related problem existed on the Python side: re-running the extraction script would append records to the raw JSONL files without checking whether a given `review_id` already existed, silently corrupting aggregated metrics.

Our implementation. We converted `fact_reviews` to an incremental model configured with a merge strategy and `review_id` as the unique key. On the first run, dbt builds the full table. On every subsequent run, an `is_incremental()` filter activates and restricts the source query to only rows whose `reviewed_at` timestamp is strictly greater than the maximum timestamp already present in the target table. The merge strategy then upserts those rows: new records are inserted, and any record whose `review_id` already exists is updated rather than duplicated.

This makes the pipeline **idempotent**: running it multiple times against the same source data produces an identical result in the fact table, a property the Lab 1 pipeline entirely lacked.

On the Python side, the `merge_reviews()` function provides a first line of defence by loading existing processed JSON, deduplicating on `review_id`, and writing back only unique records before dbt executes. This prevents duplicate rows from ever reaching the staging layer, independent of dbt's own merge logic.

2.2 Slowly Changing Dimensions — SCD Type 2

The problem. App metadata can change between scraping runs: a developer may rename their app, update their pricing, or shift categories. The original `dim_apps` table simply overwrote the previous row on each run, destroying all history of those changes and making it impossible to answer questions like *what category was this app in when these reviews were written?*

Our dbt snapshot implementation. We created a snapshot file in the `snapshots/`

folder using dbt's check strategy. The snapshot is configured with `app_id` as the unique key and a defined list of columns to monitor for changes (`app_title`, `developer_name`, `genre`, `avg_rating`, `install_count`, `price`). On the first `dbt snapshot` run, dbt inserts all rows and stamps each with `dbt_valid_from` set to the current timestamp and `dbt_valid_to` set to `NULL`. On every subsequent run, dbt compares the monitored columns against the snapshot. If any value has changed for a given `app_id`, the existing row is closed by setting `dbt_valid_to` to the current timestamp, and a new row is inserted with `dbt_valid_to = NULL`. No custom merge code is required.

A downstream model, `dim_apps_scd`, reads from the snapshot table and adds an `is_current` boolean flag (true where `dbt_valid_to` is null) for easier querying of the current state.

We validated the behaviour by manually modifying the `genre` field of one application in the source JSON and re-running the snapshot. The resulting table contained two rows for that app: one historical row with a closed `dbt_valid_to` timestamp, and one current row with `dbt_valid_to = NULL`, confirming correct SCD 2 semantics.

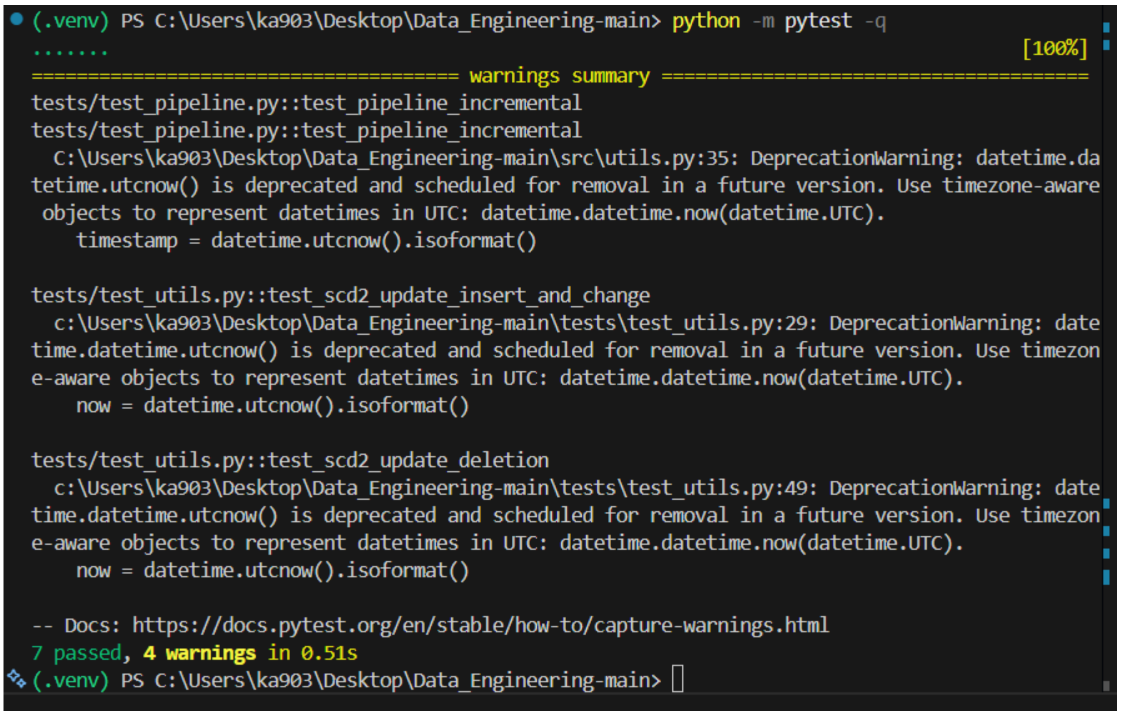
To support true historical analysis, reviews can be joined to the version of the app that was valid at review time using a temporal validity predicate: each review's `reviewed_at` timestamp is checked to fall between `valid_from` and `COALESCE(valid_to, '9999-12-31')` of the matching app record.

2.3 Data Quality and Testing

Our quality strategy operates at two independent levels: a Python pre-flight check before data reaches DuckDB, and dbt schema tests enforced at every layer of the pipeline.

Python quality module. Before any record is written to the processed folder, `quality.py` validates three categories of issue: missing identifiers (`app_id`, `review_id`), type mismatches such as non-numeric scores or unparseable timestamps, and ratings outside the valid 1–5 range. Critical failures halt the pipeline immediately, preventing corrupt data from propagating downstream.

PyTest suite. Seven automated tests cover the utility functions in `utils.py`, the incremental merge behaviour of `merge_reviews()`, and three SCD 2 scenarios (insert of a new record, update of an existing record, and logical deletion). Figure 2 shows the full test run output.

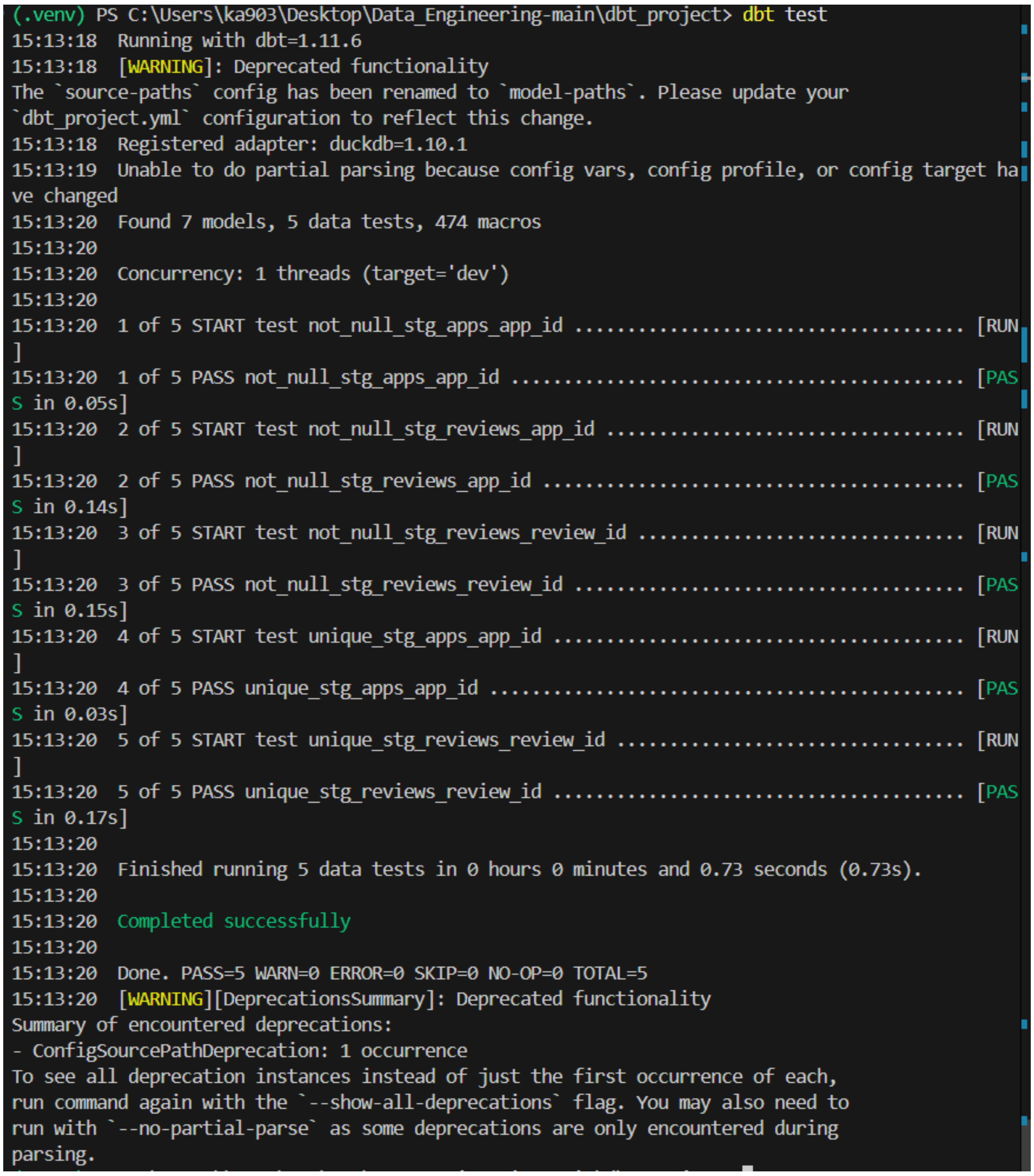


```
● (.venv) PS C:\Users\ka903\Desktop\Data_Engineering-main> python -m pytest -q
.......                                                                           [100%]
====================================== warnings summary ======================================
tests/test_pipeline.py::test_pipeline_incremental
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 tests passed in 0.51 s with 4 deprecation warnings. The warnings originate from the use of `datetime.utcnow()` in `utils.py` and are non-blocking; they do not affect test results or pipeline correctness. The three named tests at the bottom of the output — `test_pipeline_incremental`, `test_scd2_update_insert_and_change`, and `test_scd2_update_deletion` — are the most critical, as they verify that the merge and SCD 2 logic behave correctly under realistic change scenarios before any data reaches DuckDB.

dbt schema tests. Five schema tests are declared in `schema.yml` for the staging layer, covering not-null and uniqueness constraints on `app_id` in `stg_apps` and on both `app_id` and `review_id` in `stg_reviews`, plus an accepted-values test on `rating` enforcing the 1–5 range. These tests run automatically on every `dbt test` invocation and block downstream model execution if any constraint is violated.

Figure 3 shows all five tests passing.



```
(.venv) PS C:\Users\ka903\Desktop\Data_Engineering-main\dbt_project> dbt test
15:13:18  Running with dbt=1.11.6
15:13:18  [WARNING]: Deprecated functionality
The `source-paths` config has been renamed to `model-paths`. Please update your
`dbt_project.yml` configuration to reflect this change.
15:13:18  Registered adapter: duckdb=1.10.1
15:13:19  Unable to do partial parsing because config vars, config profile, or config target ha
ve changed
15:13:20  Found 7 models, 5 data tests, 474 macros
15:13:20
15:13:20  Concurrency: 1 threads (target='dev')
15:13:20
15:13:20  1 of 5 START test not_null_stg_apps_app_id ................................... [RUN
]
15:13:20  1 of 5 PASS not_null_stg_apps_app_id ......................................... [PAS
S in 0.05s]
15:13:20  2 of 5 START test not_null_stg_reviews_app_id ................................ [RUN
]
15:13:20  2 of 5 PASS not_null_stg_reviews_app_id ...................................... [PAS
S in 0.14s]
15:13:20  3 of 5 START test not_null_stg_reviews_review_id ............................. [RUN
]
15:13:20  3 of 5 PASS not_null_stg_reviews_review_id ................................... [PAS
S in 0.15s]
15:13:20  4 of 5 START test unique_stg_apps_app_id ..................................... [RUN
]
15:13:20  4 of 5 PASS unique_stg_apps_app_id ........................................... [PAS
S in 0.03s]
15:13:20  5 of 5 START test unique_stg_reviews_review_id ............................... [RUN
]
15:13:20  5 of 5 PASS unique_stg_reviews_review_id ..................................... [PAS
S in 0.17s]
15:13:20
15:13:20  Finished running 5 data tests in 0 hours 0 minutes and 0.73 seconds (0.73s).
15:13:20
15:13:20  Completed successfully
15:13:20
15:13:20  Done. PASS=5 WARN=0 ERROR=0 SKIP=0 NO-OP=0 TOTAL=5
15:13:20  [WARNING][DeprecationsSummary]: Deprecated functionality
Summary of encountered deprecations:
- ConfigSourcePathDeprecation: 1 occurrence
To see all deprecation instances instead of just the first occurrence of each,
run command again with the `--show-all-deprecations` flag. You may also need to
run with `--no-partial-parse` as some deprecations are only encountered during
parsing.
```

Figure 3: `dbt test` output — all 5 tests passed in 0.73 s with `PASS=5 WARN=0 ERROR=0`. The test names confirm the coverage: `not_null_stg_apps_app_id`, `not_null_stg_reviews_app_id`, `not_null_stg_reviews_review_id`, `unique_stg_apps_app_id`, and `unique_stg_reviews_review_id`. These results confirm that the staging layer produces clean, deduplicated, and structurally sound data before any dimensional model is built on top of it. The deprecation warning at the bottom relates to the `source-paths` config rename in dbt 1.11 and does not affect test correctness.

Dirty and drifted data handling. The staging model `stg_reviews` handles the chaos CSV files without any conditional branching or separate ingestion paths. Column name variants are resolved using `COALESCE`: the model accepts both `score` and `rating` for the star rating field, and both `at` and `review_time` for the timestamp, covering the renamed columns in `note_taking_ai_reviews_schema_drift.csv`. Invalid rating values — the string "five" and the negative value -1 from note_taking_ai_reviews_dirty.csv — are silently dropped by a TRY_CAST combined with a BETWEEN 1 AND 5 filter in the WHERE clause.

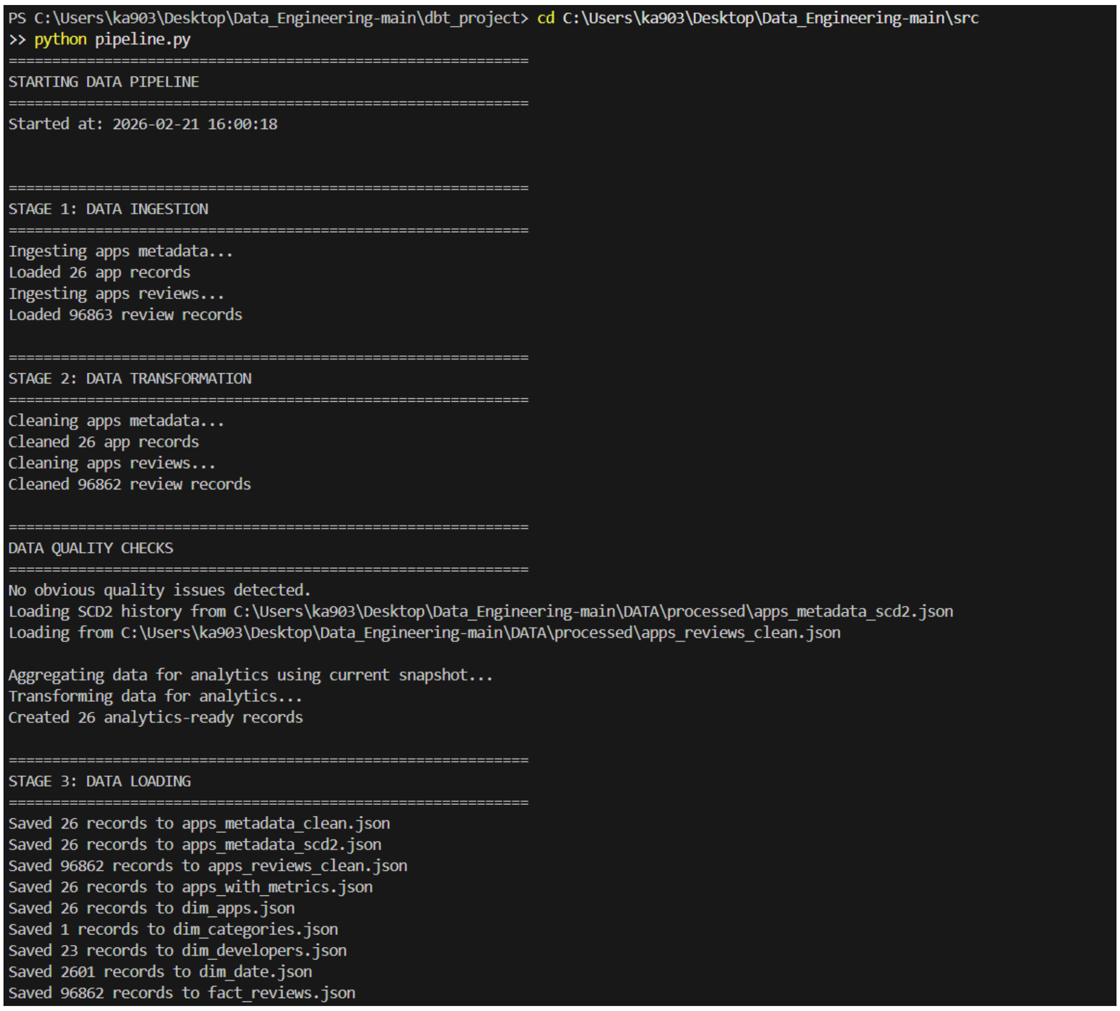
3 Python-Only vs. dbt-Based Pipeline Comparison

Table 1 compares the two architectures across six engineering dimensions, with specific references to the models and files from each lab.

Table 1: Python-only (Lab 1) vs. dbt-based (Lab 2) pipeline comparison

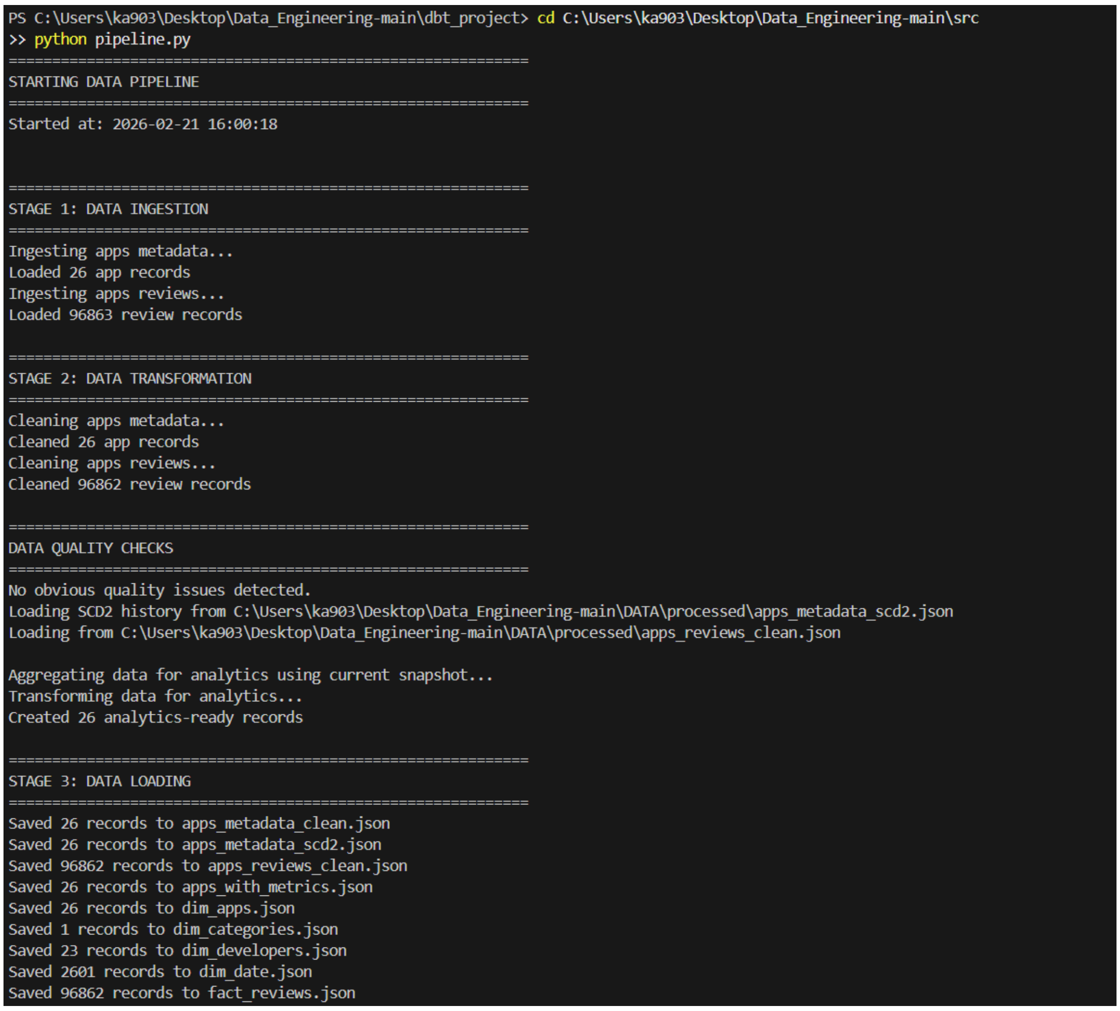
Dimension	Lab 1 — Python only	Lab 2 — dbt + DuckDB
Reproducibility	Re-running `pipeline.py` appended duplicate records to the processed JSON files, corrupting aggregated review metrics on each subsequent run	`dbt run` is fully idempotent. The incremental merge on `review_id` guarantees identical output regardless of how many times the pipeline is executed
Data quality	Quality checks existed in `quality.py` but were only enforced if the developer explicitly called that module; silent failures were possible and went undetected	Schema tests declared in `schema.yml` run automatically on every `dbt test` call. A constraint failure blocks downstream model execution, making data issues impossible to ignore
Schema drift	Hard-coded field names in `ingest.py` raised `KeyError` exceptions when column names changed. A silent `try/except` block meant those records were dropped with no operator warning	`COALESCE` expressions in `stg_reviews` accept both old and new column naming conventions in a single model, with no branching logic required
Scalability	The `merge_reviews()` function loaded all 96 862 processed records into memory on every run to perform deduplication, a pattern that fails at larger scales	The incremental filter restricts the DuckDB query to only new rows, executing inside the database engine with no Python-side memory overhead
Historical tracking	SCD 2 was implemented as custom Python logic in `utils.py`, with manual management of `start_date`, `end_date`, and `current_flag` for every tracked field	A dbt snapshot declaratively defines which columns to monitor. dbt handles the row expiry and insertion logic automatically; no custom merge code is needed
Modularity	Ingestion, transformation, and loading were tightly coupled inside `pipeline.py`. Changing a single field mapping required tracing side effects across multiple functions	Each dbt model has one responsibility. Changes propagate only through declared `ref()` dependencies, so the impact of any modification is immediately visible from the model graph

Figures 4 and 5 show the Lab 1 Python pipeline running end-to-end.



```
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 4: Python pipeline execution (`python pipeline.py`), showing all three stages completing on 2026-02-21. Stage 1 loads 26 app records and 96 863 raw review records from JSON. Stage 2 produces 96 862 clean records after removing one malformed row. Stage 3 writes nine output files to `DATA/processed/`, including the dimensional JSON exports (`dim_apps.json`, `dim_categories.json`, `dim_developers.json`, `dim_date.json`, `fact_reviews.json`) that served as the reference design for the dbt star schema in Lab 2.



```
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 5: Detailed stage output from the same Python pipeline run, highlighting the SCD 2 history file load (`apps_metadata_scd2.json`), the quality check result (*No obvious quality issues detected*), and the 26 analytics-ready records produced after joining dimension attributes to the aggregated review metrics. Comparing the 26 analytics records here with the 96 862-row `fact_reviews` table produced by dbt illustrates the key architectural difference: the Python pipeline pre-aggregated before writing, while the dbt pipeline preserves the declared grain and delegates all aggregation to the BI layer.

4 Reflections

4.1 Most Fragile Part of the Pipeline

The most fragile part was the **deduplication logic in the Python extraction layer**. In Lab 1, re-running `extract_data.py` appended new records to the same JSONL files without checking for existing `review_ids`. The `merge_reviews()` function added in Lab 2 addressed this, but by loading the entire processed JSON into memory to perform the comparison — a pattern that trades one fragility (duplicates) for another (memory exhaustion at scale).

An equally fragile point was the hard-coded field mapping in `ingest.py`. When `note_taking_ai_review` arrived with `rating` instead of `score`, the ingestion code raised a `KeyError` that was silently caught by a `try/except` block. The records were dropped entirely, with no warning logged to the operator. This is the most dangerous class of pipeline failure: silent data loss that does not raise an error.

In the dbt architecture, both fragilities are addressed structurally rather than defensively. The incremental merge on `review_id` at the database layer eliminates duplication without loading data into Python memory, and the `COALESCE` pattern in `stg_reviews` handles known column name variants transparently. However, one fragility remains: a column name we did not anticipate and did not include in a `COALESCE` expression would still silently resolve to `NULL` in the staging model, with no test catching it unless we add an explicit not-null constraint for that column.

4.2 Biggest Architectural Insight

The most valuable insight was understanding that **declaring the grain before writing any code is the highest-leverage decision in dimensional modelling**.

In Lab 1, we never formally defined what one row in the output represented. We joined apps to reviews, aggregated on whatever grouping seemed useful at the time, and wrote the result to JSON. This meant the effective grain of the output changed depending on which fields were selected, and any analyst wanting a different view had to re-derive the join logic from scratch. The 26 analytics-ready records in the Python output (as visible in Figure 5) are pre-aggregated per app — useful for one specific question, but fixed.

Once we declared the grain as one row per review event and built the Bus Matrix, every subsequent modelling decision followed logically. The Bus Matrix revealed that `dim_date` needed to be a conformed dimension — not a column derived inline — because time-based groupings must be consistent across any future fact table in the warehouse. It made the snowflake hierarchy between `dim_apps` and `dim_categories` explicit, rather than something that different analysts might join differently. And it produced the `fact_reviews` table at full granularity with 96 862 rows, which any BI tool can aggregate at query time to answer questions the Lab 1 pipeline could not support at all.

More broadly, the insight is that **a star schema is a form of documentation**. It communicates exactly what questions the warehouse can answer and at what level of detail, without requiring any reader to trace through application code to understand the structure.

4.3 One Design Decision We Would Change

We would change **how surrogate keys are generated**. All surrogate keys in our pipeline — `app_sk`, `developer_sk`, `category_sk`, and `review_sk` — are MD5 hashes of one or more natural key columns, computed independently in each model that needs them.

This approach has a practical convenience: the same surrogate key value is reproducible in any model without an explicit lookup join to the dimension table. A staging model and a fact model can both compute `MD5(LOWER(TRIM(app_id)))` and arrive at the same `app_sk`, eliminating a join. This is why we chose it.

However, MD5-based surrogate keys have two meaningful problems for a production warehouse. First, hash collisions are theoretically possible and become practically relevant as dataset size grows into the tens of millions of rows. Second, the keys are meaningless to a human inspecting a row in the fact table — a 32-character hex string conveys nothing about the app it identifies.

The Kimball recommendation is to use **integer sequence surrogate keys** generated once in the dimension table and propagated to the fact table via natural key joins. Integer keys are collision-free, human-readable in the context of an integer sequence, and substantially reduce the storage footprint of the fact table — a 4-byte integer versus a 32-byte hash string per foreign key, multiplied across 96 862 rows and four dimension keys. The reason we did not adopt this pattern is that DuckDB sequences add complexity to incremental models, because the sequence must persist correctly across multiple dbt runs. That complexity is entirely worth accepting in a production system, and it is the one aspect of our design we would revisit.

5 Conclusion

These two labs progressed from an ad-hoc Python pipeline to a modular, tested, and analytically rigorous architecture. The most significant improvements introduced in Lab 2 were reproducibility through idempotent incremental models, structural data quality through dbt schema tests that cannot be bypassed, and historical dimension tracking through dbt snapshots — all achieved with substantially less custom code than the equivalent Python implementation.

The process of applying the Kimball methodology — declaring a grain, building a Bus Matrix, and designing conformed dimensions before writing a single SQL statement — proved to be the highest-leverage activity of the entire project. It transformed an exploratory data joining exercise into a structured, documented analytical system that any BI tool can consume without additional transformation, and that any engineer can understand without reading application code.