DATA ENGINEERING

Lab 1 & Lab 2 — Full Report

Python Data Pipeline | dbt & DuckDB

Yassine TAMIM

February 2026

Prof. Lamia Ben Hiba — Centrale Casablanca

LAB 1

Python Data Pipeline — Google Play Store

A. Environment Setup

A Python 3.10 virtual environment was created using conda (environment name: **dataApps**). The following packages were installed:

- `google-play-scraper 1.2.7` — Google Play Store API access
- `pandas 2.2.0` — data manipulation and CSV output
- `plotly 5.18.0` — interactive HTML dashboard generation
- `scipy 1.11.4` — statistical analysis (linear regression for trend line)

B. End-to-End Python Pipeline

B.1 Data Ingestion (01_ingest_data.py)

Search strategy: Four search terms used — *"AI note"*, *"AI notes"*, *"note taking AI"*, *"smart notes"* — top 20 results each with deduplication via Python set.

Technical challenges:

- The `reviews_all()` function raised *'int' object has no attribute 'value'*. Switched to `reviews()` with explicit count parameter.
- Datetime serialization: scraper returns Python `datetime` objects not JSON-serializable. Custom `convert_datetime_to_string()` recursively converts to ISO strings.
- Rate limiting: 1-second delays between requests to avoid HTTP 429 errors.

Ingestion Results	42 apps extracted | 1,436 reviews collected Date range: April 2019 — February 2026 Output: data/raw/apps_catalog.json + data/raw/apps_reviews.jsonl

B.2 Data Transformation (02_transform_data.py)

Five data quality issues were identified by inspecting the raw files before writing transformation code:

#	Issue Found	Example	Fix Applied
1	Nested timestamps in 3 formats	String / dict with $date / Python datetime	Custom parse_review_date() with 3-branch logic
2	Install counts as strings	"1,000,000+" instead of integer	regexp strip '+' and ',' then cast to int

#	Issue Found	Example	Fix Applied
3	Missing values in optional fields	NULL developer names, empty content	fillna() with defaults ('Unknown', '')
4	Price with currency symbols	"$4.99" instead of float	re.sub to strip non-numeric chars
5	Inconsistent genre type	String or list depending on app	isinstance() check + str() conversion

Transformation Results	Apps: 42 → 42 clean records (no data loss) Reviews: 1,436 → 1,436 clean records Output: data/processed/apps_catalog.csv + data/processed/apps_reviews.csv

B.3 Serving Layer (03_create_serving_layer.py)

Two analytics-ready aggregated datasets were produced:

- **app_level_kpis.csv**: one row per app — review count, average rating, % low ratings (score ≤ 2), first/last review dates. 38 of 42 apps had at least one review.
- **daily_metrics.csv**: one row per day — daily volume and daily average rating. 415 unique days tracked across the full date range.

B.4 Dashboard (04_create_dashboard.py)

An interactive HTML dashboard was built with Plotly using a 3×3 subplot grid. Theme: `plotly_white` with custom dark-blue palette.

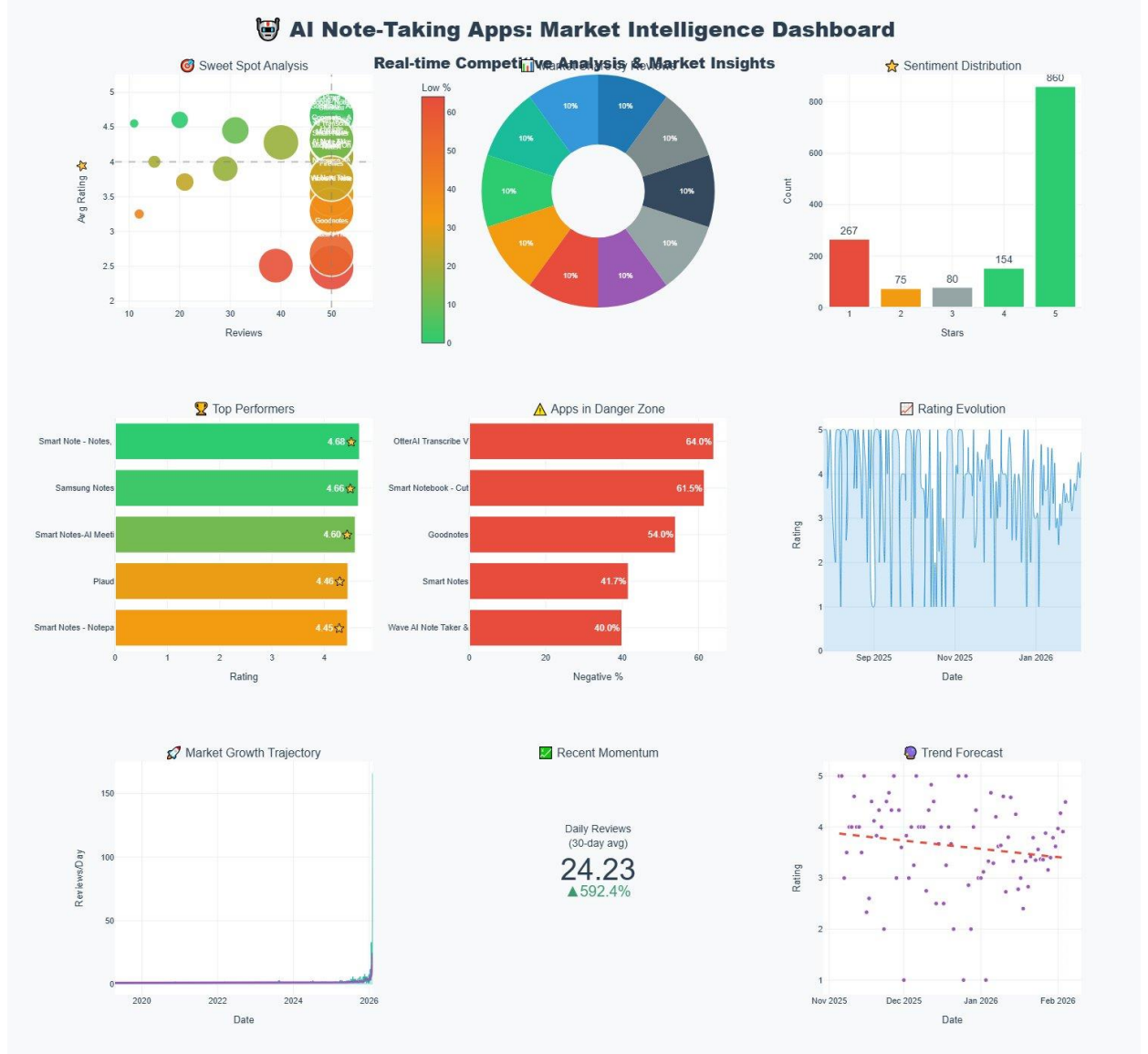

Figure 1 — Lab 1 Dashboard: AI Note-Taking Apps Market Intelligence (9 visualizations)

Key Findings from the Dashboard

Category	App	Rating	Key Metric
Top Performer	Smart Note — Notes	4.68 ★	Highest rated in dataset
Top Performer	Samsung Notes	4.66 ★	Established player holding strong
Top Performer	Smart Notes-AI Meeting	4.60 ★	—
Danger Zone	OtterAI Transcribe V	2.48 ★	64.0% negative reviews — worst in dataset
Danger Zone	Smart Notebook - Cut	2.51 ★	61.5% negative reviews
Danger Zone	Goodnotes	2.68 ★	54.0% negative reviews

Market Trend: Review volume spiked +592% in late 2025 (24.23 reviews/day 30-day avg). Average rating trended DOWN from 5.0 to ~3.5 over the observation period — user expectations are rising faster than product quality. The sentiment distribution shows 860 five-

star vs 267 one-star reviews, but the negative tail is significant and concentrated in specific apps.

Rating distribution: 267 × ★1 | 75 × ★2 | 80 × ★3 | 154 × ★4 | 860 × ★5 — bimodal distribution (love-it or hate-it market)

C. Pipeline Pain Points

Five structural weaknesses were identified that motivated the Lab 2 re-engineering:

Pain Point	What Breaks	Lab 1 Symptom
No schema validation	Google changes API field name from 'score' to 'rating'	Silent failure — all ratings become NULL, dashboard shows garbage
No incremental updates	Must re-scrape ALL 1,436 reviews every run	10-minute runtime for what should be a 30-second delta update
Hardcoded everything	App IDs and file paths in source code	Changing the app list requires editing Python, not a config file
Zero observability	Pipeline fails silently midway	No logging — failures only discovered when someone checks the dashboard
Cannot scale	Pandas loads entire dataset into RAM	Works at 1,436 rows; breaks at 1,000,000 rows on a laptop

Stress Test	"What if Google changes the API tomorrow?" → The pipeline fails silently or produces NULL-filled CSVs. → The dashboard shows incorrect data with no error visible. → There is no rollback mechanism, no alerting, and no way to recover the last good state. → This is exactly why production pipelines need orchestration, schema management, and testing frameworks.

LAB 2

Data Pipeline with dbt & DuckDB

A. Environment Setup

Lab 2 reuses the same conda environment (**dataApps**, Python 3.13.2). Three packages added:

- `dbt-core==1.7.4` — open-source SQL transformation framework
- `dbt-duckdb==1.7.2` — DuckDB adapter for dbt
- `duckdb==0.9.2` — columnar analytical execution engine

B. High-Level Architecture

The Python scraper is retained for generation and ingestion. dbt replaces all ad-hoc pandas transformation logic. DuckDB serves as both the execution engine and analytical storage.

Layer	Tool	Responsibility	Output
Generation	Python / google-play-scraper	Scrape apps and reviews from Google Play API (unchanged from Lab 1)	apps_catalog.json, apps_reviews.jsonl
Ingestion	Python load_to_duckdb.py	Bridge Lab 1 CSVs into DuckDB raw schema — idempotent batch append via _source_file guard	raw.apps_catalog, raw.apps_reviews
Transformation	dbt Core (SQL models)	Staging cleanup → dimensional modeling → incremental fact table — explicit DAG via ref()	stg_*, dim_*, fact_reviews
Serving	DuckDB / Power BI / Metabase	Analytics-ready star schema queryable by any BI tool supporting DuckDB or ODBC	marts schema in app_market.duckdb

C. Data Modeling — Kimball Methodology

Grain Declaration

Grain	One row in the fact table represents one review submitted by one user for one app on one specific date. Natural key: reviewId (globally unique per review, provided by the Google Play API). This is the finest grain available — no aggregation occurs in the fact table itself.

Dimensions, Facts & Bus Matrix

Dimension	Business Meaning	Source	Key Attributes
dim_apps	The application being reviewed	stg_apps (apps_catalog.csv)	app_key (SK), app_id (NK), app_name, price, is_paid, installs, catalog_rating
dim_developers	Developer of the app, tracked historically via SCD2	stg_apps + scd2_developers snapshot	developer_key (SK), developer_name, valid_from, valid_to, is_current
dim_categories	Genre / category of the app	stg_apps	category_key (SK), category_name
dim_date	When the review was submitted	Generated SQL spine	date_key (YYYYMMDD int), year, month, quarter, day_of_week, is_weekend

Fact \ Dimension	dim_apps	dim_developers	dim_categories	dim_date
fact_reviews	X	X	X (via dim_apps)	X

D. dbt + DuckDB Pipeline

D.1 Configuration

Project initialized in `dbt/` subfolder. DuckDB path: `../data/app_market.duckdb`. Materialization: staging = `view`, dimensions = `table`, fact = `incremental`.

D.2 Staging Layer

stg_apps

- `TRIM(appId)` → whitespace on natural key; `regexp_replace(installs,'[^0-9]','')` → strips '1,000,000+' to bigint
- `coalesce(genre,'Unknown')` → null-safe category; `ROW_NUMBER() OVER (PARTITION BY app_id ORDER BY _loaded_at DESC)` → deduplication

stg_reviews

- Column names match Lab 1 output exactly: `reviewId`, `score`, `content`, `thumbsUpCount`, `at`, `userName`
- Quality gate: drops rows where rating ∉ [1-5], timestamp is NULL, or date before 2015-01-01
- `ROW_NUMBER() OVER (PARTITION BY review_id ORDER BY _loaded_at ASC)` → deduplication across batches

D.3 Dimension Tables & Actual Schema

After dbt build, the DuckDB catalog confirmed the following table schemas:

Model	Type	Key Columns (from catalog.json)
dim_apps	BASE TABLE	app_key VARCHAR, app_id VARCHAR, app_name VARCHAR, developer_key VARCHAR, category_key VARCHAR, price DOUBLE, is_paid BOOLEAN, installs BIGINT, catalog_rating DOUBLE
dim_developers	BASE TABLE	developer_key VARCHAR, developer_name VARCHAR, valid_from TIMESTAMP, valid_to TIMESTAMP, is_current BOOLEAN
dim_categories	BASE TABLE	category_key VARCHAR, category_name VARCHAR
dim_date	BASE TABLE	date_key INTEGER, date DATE, year INT, month INT, quarter INT, day_of_week INT, is_weekend BOOLEAN, month_name VARCHAR, year_month VARCHAR
fact_reviews	BASE TABLE	review_id VARCHAR, app_key VARCHAR, developer_key VARCHAR, date_key INTEGER, rating INTEGER, thumbs_up_count INTEGER, review_text VARCHAR, _loaded_at VARCHAR, _source_file VARCHAR
stg_apps	VIEW	app_id, app_name, developer_name, catalog_rating DOUBLE, ratings_count INT, installs BIGINT, category_name, price DOUBLE, is_paid BOOLEAN, _loaded_at
stg_reviews	VIEW	review_id, app_id, rating INTEGER, review_text, thumbs_up_count INT, reviewed_at TIMESTAMP, user_name, _loaded_at, _source_file
scd2_developers (snapshot)	BASE TABLE	developer_name, updated_at, dbt_scd_id, dbt_updated_at TIMESTAMP, dbt_valid_from TIMESTAMP, dbt_valid_to TIMESTAMP

D.4 Fact Table — fact_reviews

Materialized as `incremental` with `unique_key='review_id'` and `incremental_strategy='delete+insert'`. INNER JOINs to dim_apps and dim_date enforce referential integrity — reviews for unknown apps or invalid dates are excluded by the staging quality gate before reaching this model.

Incremental Filter Logic	First run: full load — {% if is_incremental() %} block is skipped. Subsequent runs: WHERE _loaded_at > (SELECT MAX(_loaded_at) FROM this) delete+insert on unique_key ensures idempotency — same batch run twice = same result.

D.5 dbt Build Results — 37 Tests, 0 Errors

Test Category	Count	Coverage
not_null / unique (generic)	24	All PK/SK/FK columns across all 5 models
Relationship / FK integrity	6	fact_reviews → dim_apps, dim_date, dim_developers; dim_apps → dim_categories, dim_developers
accepted_values	4	is_paid ∈ {true,false}, is_current ∈ {true,false}

Test Category	Count	Coverage
Custom SQL tests	3	assert_no_orphan_reviews, assert_rating_distribution_sane, assert_no_data_loss

D.6 dbt Lineage Graph

The following lineage graph (generated by dbt docs serve) shows the full DAG from raw sources through staging, snapshot, dimensions, to fact table and custom tests:

Figure 2 — dbt Lineage Graph: raw sources (green) → staging → snapshot/dimensions → fact_reviews → custom tests

The graph confirms the correct dependency order: `raw.apps_catalog` feeds both `stg_apps` and the `scd2_developers` snapshot in parallel. `stg_reviews` and the dimensions converge at `fact_reviews`, which then feeds all three custom data quality tests.

E. Chaos Engineering

E.1 Incremental Loading

Ingestion layer: load_to_duckdb.py stamps each row with `_loaded_at` and `_source_file`. Before appending any batch, it checks if that filename is already present in raw.apps_reviews — if so it prints `[SKIP]`, making every run idempotent.

dbt layer: fact_reviews processes only rows where `_loaded_at > MAX(_loaded_at)` in the existing table. Combined with delete+insert, the model is fully idempotent.

E.2 SCD Type 2 — Developer Dimension

Debug Note	First attempt: snapshot used {{ ref('stg_apps') }} — immediately failed with 'Table stg_apps does not exist'. Root cause: dbt snapshots run BEFORE dbt build. Staging views don't exist at snapshot time. Fix: snapshot reads directly from {{ source('raw', 'apps_catalog') }} — the raw DuckDB table. This is a real dbt execution-order constraint, discovered during implementation.

The scd2_developers snapshot uses the check strategy on developer_name. On first run: every developer gets one open-ended record (dbt_valid_to = NULL). On subsequent runs: if a name changes, the old row gets dbt_valid_to stamped, and a new row is inserted. The catalog.json confirms the snapshot schema: dbt_valid_from TIMESTAMP, dbt_valid_to TIMESTAMP, dbt_scd_id VARCHAR.

F. Python-Only vs dbt-Based Pipeline

Dimension	Lab 1 (Python + pandas)	Lab 2 (dbt + DuckDB)
Incremental Loading	Full re-scrape every run — re-processes all 1,436 rows.	Incremental model: only rows with _loaded_at > last run. Loader skips already-ingested files.
Schema Drift	Hard crash requiring edits across multiple scripts.	load_to_duckdb.py normalizes at ingestion. Downstream SQL unchanged.
Deduplication	drop_duplicates() once — no guarantee across separate batch runs.	ROW_NUMBER() in stg_reviews + unique_key on fact_reviews = enforced at every layer.
Testing	Manual print statements and visual CSV inspection. Silent failures.	37 automated tests. Build fails loudly before bad data reaches the serving layer.
Historical Tracking	No history. Developer renames silently overwrite previous records.	SCD2 snapshot: full history preserved. Reviews linked to correct developer version at review time.
Dependency Order	Hardcoded 01_/02_/03_/04_ script naming. Manually maintained.	dbt ref() builds explicit DAG. Correct order automatically enforced every build.
Scalability	Pandas in RAM. Breaks at ~1M rows on a laptop.	DuckDB: columnar, handles 100M+ rows. Change one line in profiles.yml to scale to cloud.
Reproducibility	Depends on run order and local file state.	dbt build: fully reproducible — same inputs always produce same outputs.

G. Reflections

Most Fragile Part of the Pipeline

The implicit contract between `load_to_duckdb.py` and `stg_reviews.sql`. The loader must produce column names matching exactly what Lab 1's `02_transform_data.py` writes to CSV. This caused a real bug during development: stg_reviews used COALESCE alias chains for schema drift resilience, but this created a silent SKIP when dbt evaluated it against a table using Lab 1's exact column names only. The fix simplified stg_reviews to use exact names (`reviewId`, `score`, `thumbsUpCount`). The contract is now documented in comments but still implicit — this boundary remains the most fragile point.

Biggest Architectural Insight

dbt's separation of concerns is **structurally enforced**, not just conventional. In Lab 1, `02_transform_data.py` simultaneously reads raw JSON, cleans types, joins app names, computes KPIs, and writes CSV — all in one function. A schema change at the raw layer breaks all of those responsibilities at once. In Lab 2, a raw schema change requires editing exactly **one file** (`stg_reviews.sql`) and zero downstream models change. This loose coupling was described in lectures as a principle; building both pipelines made it concrete, measurable, and viscerally real when the bug hit during development.

One Design Decision I Would Change

I would not implement SCD2 on dim_developers in its current form. The snapshot tracks `developer_name` changes, but the Google Play scraper exposes no `developer_website` or `developer_email` — those columns are permanently NULL. The SCD2 machinery (snapshot running before build, `WHERE is_current = true` filters everywhere, deeper DAG) adds complexity before delivering any analytical value a simple Type-1 dimension could not provide. The correct approach: build dim_developers as Type-1 now, migrate to SCD2 only when a developer profile enrichment step is added that makes the historical tracking genuinely meaningful.

Appendix — Final Repository Structure

```
App_Market_Research/
├── src/                              Lab 1 pipeline (unchanged)
│   ├── 01_ingest_data.py
│   ├── 02_transform_data.py
│   ├── 03_create_serving_layer.py
│   └── 04_create_dashboard.py
├── data/
│   ├── raw/                          Immutable scraped files
│   ├── processed/                    Lab 1 clean CSVs
│   └── app_market.duckdb             Lab 2 DuckDB database
├── dbt/
│   ├── dbt_project.yml
│   ├── profiles.yml
│   ├── models/staging/               stg_apps.sql, stg_reviews.sql, sources.yml
│   ├── models/marts/                 dim_*.sql, fact_reviews.sql, schema.yml
│   ├── snapshots/scd2_developers.sql
│   └── tests/                        3 custom SQL tests
├── scripts/load_to_duckdb.py         Idempotent CSV-to-DuckDB loader
├── run_pipeline.py                   Lab 1 runner (unchanged)
└── requirements.txt
```