Lab 1

Python-only Data Pipeline
January 2026

Objectives

- Install and Set up our environment
- Understand the stages of the Data Lifecycle
- Implement an End-to-End Data Pipeline
- Experience typical pain points of built-by-hand Pipelines

A. Environment Setup

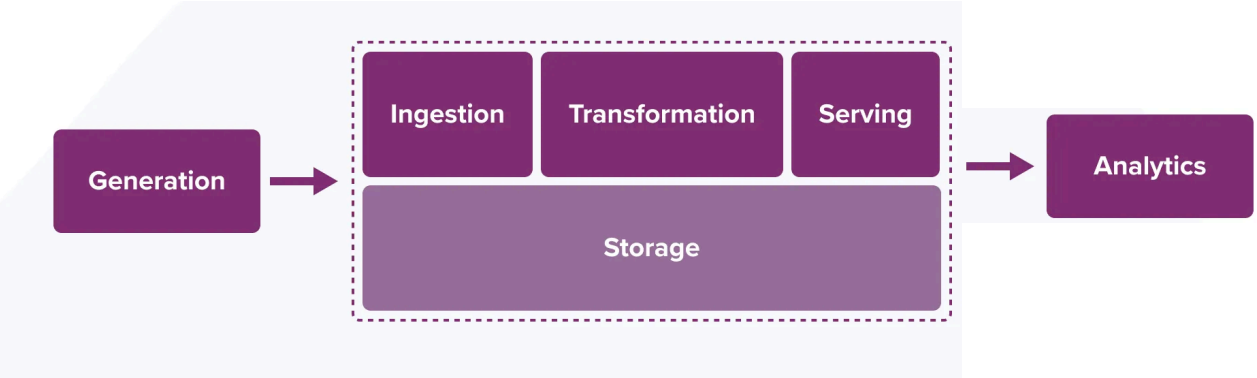
For this first lab, we will create a Python-only Data Pipeline. We will need a Python environment 3.7+ (It is recommended to create a **dedicated virtual environment** for these labs). It is best to use an IDE rather than Jupyter notebooks for this lab.

✓ Start by creating a virtual environment where we will install all needed libraries as we go.

When running your code, make sure you are running it in the right virtual environment (e.g. in the terminal you need to conda activate your environment, in visual Code IDE you can choose your environment by clicking on the bottom right banner).

B. End-to-End Data Pipeline

For the purpose of this lab, we will be acting as a junior data engineer for a product analytics team. Our task is to build a Python-only pipeline that converts raw data into analytics-ready outputs. A data pipeline is the combination of architecture, systems, and processes that move data through the stages of the data engineering lifecycle (as illustrated in the figure below).



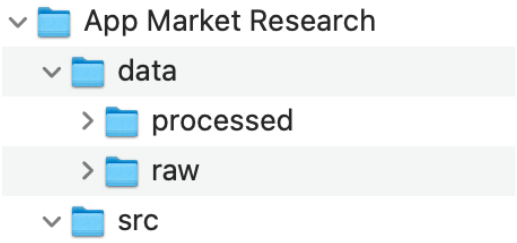
For the purpose of competitive market research, our company wants to track AI note-taking mobile apps.

We have two datasets:

- Apps Catalog
  - Metadata about each AI note-taking app
  - Serves as the “reference / dimension” dataset

- Apps Reviews
  - User reviews posted in app stores
  - Serves as the “events / fact” dataset

We will set up our project with a very simple structure:



The **data>raw** contains the original datasets and represents the output of the “Generation” stage in the data lifecycle. After cleaning, transforming, and aggregating our data, we will save our files in the **data>processed** folder. The **src** folder contains the pipeline logic (Python scripts) responsible for reading raw data, transforming it, and writing processed outputs.

1. Data Acquisition and Ingestion

The data acquisition step is the entry point to the data pipeline. It can encompass creation and/or capture of data generated from different sources. Data can be ingested from flat files, source databases, APIs etc.

While working with databases is quite straightforward seen that the data is structured. Many data applications use a mixture of structured, semi-structured, and/or unstructured data. To explore the challenges of less structured data, we will build a pipeline based on semi-structured data.

The automated gathering of data from the web is a practice that is increasingly used by companies to gain competitive advantage. For instance, web data can help a business conduct market research and study its competitors more effectively or keep up to speed on shifting industry trends.

The process of extracting data from websites is called **web scraping** (also web harvesting). It consists of browsing websites and collecting specific, relevant data points for our business needs. Web scrapers are excellent at gathering and processing large amounts of data quickly but it can get easily messy. That is why, when available, we generally use **APIs** to extract the data. An API (Application Programming Interface), as the name suggests, is an interface allowing one system/application to interact with another. Practically, it gives an application a set of functions and procedures that allow for the interaction with the source system, including the extraction of data etc. APIs offer more structured data results, but might be limited (e.g. number of requests per day…).

For the sake of our use case, we will be using **Google's Play Store** as a data source. Google Play store is a digital distribution service operated and developed by Google. It serves as the official app store for certified devices running on the Android operating system and its derivatives allowing users to browse and download applications developed with the Android software development kit (SDK) and published through Google. Recent estimates place the number of apps on the platform around 3.5 to 4.2 million apps.

Since we need data on our competitors' applications, we will extract data on available apps using Google Play API. We can use the API offered by Google, or as it is often the case for Python, use a library that facilitate data extraction, for example: Google-Play-Scraper.

- ✓ In your virtual environment, install the Google Play Scraper library of your choice
- ✓ Explore data that can be extracted using the API (through the documentation of the chosen library and snippets of codes you can try)
- ✓ Extract as much data as you can on note taking AI applications (the more data you can get, the better for later labs). We would like not only to have data on the applications but reviews made by users of those apps.

✓ Store your raw data **(as is)** in JSON (or JSONL) files. Create a file for the apps metadata and another one for the reviews. Don't apply any transformations yet.

2. Diagnosing and Transforming Raw Data

In this step, we will examine the raw JSON/JSONL datasets and identify the issues that prevent them from being used directly for analytics. In this step, we will transform the raw data into structured, usable datasets. The goal is to convert semi-structured inputs into tabular data that can be analyzed and aggregated.

✓ Before writing any transformation code, inspect the raw JSON/JSONL files. Spend time inspecting the overall structure of each file, the types of values stored in each field, the presence of nested structures, missing or inconsistent values, and fields that may not be suitable for analytics in their current form. Document your observations informally (comments in code or notes).

✓ Based on your exploration, identify at least five issues in the raw datasets that would cause problems for analytics or downstream use.

✓ Design your transformation strategy.

✓ Implement your transformation logic in Python. Write your resulting datasets to the data>processed folder.

Guidelines:

- The final structure of your files should be:
    - Apps Catalog: appId, title, developer, score, ratings, installs, genre, price
    - Apps Reviews: app_id, app_name, reviewId, userName, score, content, thumbsUpCount, at
- Transformations may be implemented in one script or multiple functions
- Intermediate datasets may be written to disk if you find it useful
- Avoid modifying raw data files directly
- Ensure your code can be re-run from scratch

✓ After transformation, verify that: datasets are tabular and consistent, key fields can be used for joins, numeric fields behave numerically, timestamps can be aggregated by day, obvious anomalies are handled or documented

3. Serving Layer

In this step, we will prepare the datasets for downstream consumers (dashboarding, reporting, analytics). Using our transformed/cleaned reviews table, we will produce two "serving layer" outputs: one at the app level and one at the daily level.

Guidelines:

- We will only use the datasets produced during transformation (e.g., from data>processed)
- Outputs must be reproducible: re-running the pipeline should regenerate the same serving files.

Output 1 — App-Level KPIs

Create a csv file in data>processed where each row corresponds to one application and contains the following metrics:

- Number of reviews
- Average rating
- % of low rating reviews (rating ≤ 2)
- First review date
- Most recent review date

Output 2 — Daily Metrics

Create a daily time series csv file where each row corresponds to one date and contains:

- Daily number of reviews
- Daily average rating

4. Lightweight Dashboarding (Consumer View)

In this final step, we will take the role of a data consumer and build a very small dashboard using the outputs of our pipeline. The goal is not to design a perfect dashboard, but to verify that our pipeline produces usable analytics data and experience the separation between data engineering and data consumption. Don't modify your pipeline during this step.

Use only the datasets produced in the previous steps, located in data>processed. You may use any simple library you are comfortable with (e.g. Plotly, Bokeh, Matplotlib, etc.)

Your dashboard should help answer at least one of the following:

✓ Which applications appear to perform best or worst according to user reviews?
✓ Are user ratings improving or declining over time?
✓ Are there noticeable differences in review volume between applications?

Briefly explain (2–3 sentences) what your dashboard shows.

C. Pipeline changes and Stress testing

In real-world data systems, pipelines rarely remain static. New data arrives under different conditions, schemas evolve independently of downstream consumers, data quality degrades in unexpected ways, and upstream systems may change the way data is delivered without coordination.

In this step, we will intentionally stress-test our data pipeline by running it against several modified versions of the upstream datasets. Each dataset provided here must be treated as a replacement of the upstream data source, rather than as an addition to previously processed data, regardless of its file format.

Although earlier steps in this lab used JSON or JSONL files as raw inputs and CSV files as processed outputs, in this step some upstream systems will provide data directly in CSV format; these files should therefore be considered *raw data relative to the pipeline boundary*. This design choice allows us to isolate how robust our pipeline is to realistic changes such as new data batches, schema drift, data quality issues, and unexpected input formats, without introducing the additional complexity of incremental state management.

The objective is not to make the pipeline perfect, but to expose hidden assumptions and structural fragilities that commonly appear in early-stage data pipelines, and to understand why more structured tools and patterns are needed in later labs.

Guidelines:

- No manual edition of any raw files
- All adaptations must be done in code

1. New Reviews Batch

In file “note_taking_ai_reviews_batch2.csv”, you are provided with a new reviews dataset representing a different batch of data produced by the upstream system. Although the dataset may appear similar to the original one, it contains subtle differences that reflect common real-world ingestion challenges.

You should run your existing pipeline using this dataset as the sole reviews input and observe the outcome. Pay particular attention to how your code handles repeated identifiers, unknown applications, or unexpected

variations in content. Consider whether your pipeline logic clearly expresses whether it is performing a full rebuild of the dataset or relying on implicit assumptions about continuity.

Tasks

- ✓ Update your pipeline so that this new batch is included.
- ✓ Re-run the pipeline from scratch.

Observe

- ✓ How many changes were required in your code to support this new batch?
- ✓ Is your pipeline clearly performing a full refresh, or does this behavior remain implicit?
- ✓ How are duplicate reviews handled?
- ✓ What happens to reviews that reference applications not present in the applications dataset?

2. Schema Drift in Reviews

In file "note_taking_ai_reviews_schema_drift.csv", you are provided with a reviews dataset in which the column names and field structure differ from what your pipeline was originally built to expect. This situation, commonly referred to as **schema drift**, frequently occurs when upstream systems evolve independently of downstream consumers.

Replace the reviews input with this dataset and execute your pipeline without modifying the raw file. Observe where your code relies on hard-coded column names, implicit schemas, or positional assumptions. Reflect on whether failures are immediately visible or whether incorrect results are produced silently, and consider how difficult it is to adapt the pipeline to this change.

Tasks

- ✓ Replace the original reviews dataset with this version.
- ✓ Re-run your pipeline without modifying raw files.
- ✓ If necessary, adapt your code to accommodate the new structure.

Observe

- ✓ Which parts of your pipeline rely on hard-coded column names?
- ✓ Does the pipeline fail explicitly, or does it produce incorrect results silently?
- ✓ How localized or widespread are the required code changes?

3. Dirty and Inconsistent Data Records

We will then work with a reviews dataset that contains invalid, missing, or inconsistent values [file "note_taking_ai_reviews_dirty.csv"]. These issues may include incorrect data types, out-of-range values, malformed timestamps, or ambiguous representations of missing data.

Run your pipeline on this dataset and carefully observe how such records are handled. Consider whether errors are detected early, whether invalid values propagate into downstream aggregates, and whether your pipeline makes explicit decisions about data quality or simply relies on default behavior. This step is designed to highlight how easily analytical results can be affected by unhandled data quality problems.

Tasks

- ✓ Run your pipeline on this dataset without cleaning the raw file manually.
- ✓ Allow the pipeline to handle invalid records according to its current logic.

Observe

- ✓ How does your pipeline handle invalid ratings or timestamps?
- ✓ Are problematic records filtered out, transformed, or propagated downstream?
- ✓ Do data quality issues surface early, or do they affect aggregated metrics silently?

4. Updated Applications Metadata

In addition to changes in the reviews data, the applications metadata has also evolved in file “note_taking_ai_apps_updated.csv”. The updated dataset introduces issues such as duplicated identifiers, missing values, and inconsistencies in numeric and categorical fields.

Replace the applications dataset with the updated version and re-run your pipeline. Observe how these changes affect joins between reviews and applications, as well as downstream aggregations. Reflect on whether your pipeline enforces assumptions about uniqueness and referential integrity, or whether such assumptions remain implicit.

Tasks

- ✓ Replace the original apps metadata with this version.
- ✓ Re-run the pipeline.

Observe

- ✓ How are duplicate application identifiers handled?
- ✓ What happens during joins between reviews and applications?
- ✓ Are downstream aggregates affected in ways that are immediately visible?

5. New Business Logic Stress Test (Consumer-Driven Change)

In practice, changes to data pipelines are not driven only by upstream data sources. Very often, new requirements originate from downstream consumers, such as analysts, product managers, or dashboard

users. Even when raw data remains unchanged, evolving business questions can require significant modifications to transformation logic, data models, and serving outputs.

In this step, we will examine how a change in analytical requirements can impact the pipeline we have built so far.

Business request:

"We want to identify applications where the sentiment expressed in review text appears to contradict the numeric rating (e.g., highly negative text paired with a high score, or positive text paired with a low score)."

Tasks

- ✓ Starting from your existing processed datasets, determine whether your current serving outputs are sufficient to answer this question.
- ✓ If not, identify which additional fields, transformations, or derived metrics would be required.
- ✓ You may implement a simple heuristic (e.g., keyword-based sentiment indicators) or limit yourself to a design proposal without full implementation.

- ✓ You are not required to modify raw data files, but you may need to:
  - ✓ extend transformation logic,
  - ✓ modify serving-layer tables,
  - ✓ or introduce new intermediate datasets.

Observe

- ✓ Where in your pipeline would this new logic naturally belong?
- ✓ How many parts of the pipeline would need to change to support this request?
- ✓ Would this logic be easy to reuse or maintain if additional business questions were introduced?
- ✓ Does your current pipeline structure clearly separate data preparation from analytical logic?