Frist Page

Content

# Introduction

In 2025, **watching movies is a widely accessible and popular activity**. Streaming platforms greatly benefit from this popularity and offer various services, such as providing online movie streaming services, managing subscription payments, and offering comprehensive customer support. One crucial service for retaining customers is **movie recommendation**. This process is driven by **data analysis** and powered by a **recommendation system** that utilizes high-quality data derived from these analytical processes.

The fundamental process that ensures high-quality data for data analytics and building models is **data engineering**. It focuses on efficient data processing and **data pipeline system maintenance**, ultimately leading to actionable, high-quality data.

This project utilizes the **TMDB (The Movie Database) dataset**, a free, community-driven platform providing detailed information about movies, TV shows, and cast members. It includes metadata such as genres, ratings, production details, and images. Developers can access this rich dataset via a public API, making it ideal for building recommendation systems and movie-related projects.

This project presents the architecture and data pipeline system utilizing tools within the **data engineering field**. For example, **Apache Airflow** for orchestrating and managing data pipelines, **Apache Spark** for processes and transforms the large TMDB dataset efficiently, and **BigQuery**, a data warehouse service from Google Cloud Platform (GCP), for storing high-quality data. This project demonstrates **the end-to-end data engineering process**. The outcome of this project is high-quality data prepared for data analytics and recommendation systems.

## Objective

1.**To design and implement a robust and scalable data architecture** that integrates all components (data ingestion, processing, storage, and serving) to support the end-to-end data pipeline.

2.**To build a data processing pipeline** that ensures high-quality data is extracted, transformed, and loaded into the data warehouse.

3.**To establish a scalable data warehouse** to house high-quality movie data, optimized to efficiently support a movie recommendation system.

## Output

The successful execution of this project, **TMDB-RecoFlow**, will yield the following key outputs:

* **A robust and scalable data pipeline**: This end-to-end pipeline, built with Apache Airflow for orchestration and Apache Spark for processing, ensures continuous data flow from ingestion to serving.
* **An optimized BigQuery data warehouse**: A well-structured central repository for high-quality, transformed movie metadata, designed for efficient data retrieval to support analytical queries and recommendation system training.
* **A high-quality dataset for recommendation**: The project will provide a clean, consistent, and readily accessible dataset specifically prepared to feed into a movie recommendation system for suggesting similar titles.
* **Demonstration of end-to-end data engineering**: This project will serve as a practical showcase of the complete data engineering lifecycle, from raw data acquisition to delivering actionable data for advanced analytics.

## Benefits

* **Enhanced Operational Efficiency**: The automated data pipeline, powered by Apache Airflow and Spark, streamlines data processing, reducing manual effort and ensuring timely data availability.
* **Actionable Business Insights**: The project delivers clean, structured data in BigQuery, enabling precise analytics and valuable insights into movie trends and user behavior.
* **Practical Skill Development**: For the developer, it provides invaluable hands-on experience in designing, implementing, and managing an end-to-end data engineering pipeline using industry-standard tools.

# Data Source

**Name:** Full TMDB Movies Dataset 2024 (1M Movies)

**Source:** <https://www.kaggle.com/datasets/asaniczka/tmdb-movies-dataset-2023-930k-movies/data>

**Data Characteristic:**

**The TMDB Movies Dataset (2023)** used in this project is a comprehensive and regularly updated collection of **film information**. It contains a vast number of movies, totaling **1,000,000 entries from the TMDB database**, with daily updates. Each entry includes essential details such as the movie's ID, Title, Average Vote, Vote Count, Status, Release Date, Revenue, and Runtime. Additionally**, the dataset features various other attributes that contribute to effective analysis and the development of robust movie recommendation systems.**

**Concern:**

* **Secondary Source**: The data is obtained from Kaggle, a secondary source, rather than directly from the TMDB API. This means the project relies on someone else's extraction and aggregation process, which might introduce unforeseen biases or limitations from their collection methodology.
* **Snapshot Nature**: The dataset is a static snapshot, containing data only up to (02-06-2025)

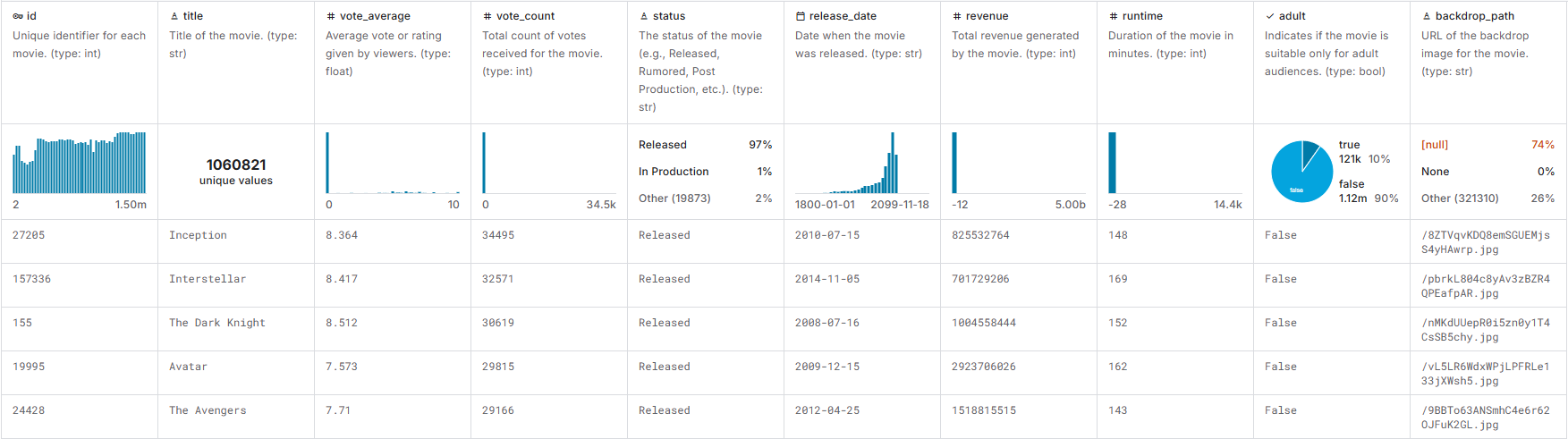
**Data Dictionary:**

The dataset exported from Kaggle contains 1,234,214 rows and 24 columns. This snapshot was created on 02‑06‑2025 (DD-MM-YYY)

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Data Type** | **Description** |
| id | int | Unique identifier for each movie |
| title | str | Title of the movie |
| vote\_average | float | Average rating given by viewers |
| vote\_count | int | Total number of votes received |
| status | str | Production status (e.g., Released, Rumored) |
| release\_date | str | Date when the movie was/will be released |
| revenue | int | Total box office revenue |
| runtime | int | Duration of the movie in minutes |
| adult | bool | Whether the movie is categorized as adult-only |
| backdrop\_path | str | URL path to the movie's backdrop image |
| budget | int | Production budget of the movie |
| homepage | str | Official homepage URL for the movie |
| imdb\_id | str | IMDb identifier |
| original\_language | str | Original language code (e.g., en, fr) |
| original\_title | str | Original title of the movie |
| overview | str | Synopsis or description |
| popularity | float | Popularity score determined by TMDB |
| poster\_path | str | URL path to the movie's poster image |
| tagline | str | Official tagline or slogan |
| genres | str/list | List of genres (as strings) |
| production\_companies | str/list | List of production companies |
| production\_countries | str/list | List of production countries |
| spoken\_languages | str/list | List of languages spoken in the movie |
| keywords | str/list | List of associated keywords |

**Column Classification in Dataset**

|  |  |  |
| --- | --- | --- |
| **Numeric** | **Categorical (incl. bool/list)** | **Other/String/ID** |
| vote\_average | status | id |
| vote\_count | adult | title |
| revenue | original\_language | release\_date |
| runtime | genres | backdrop\_path |
| budget | production\_companies | homepage |
| popularity | production\_countries | imdb\_id |
|  | spoken\_languages | original\_title |
|  | keywords | overview |
|  |  | poster\_path |
|  |  | tagline |

****

**Figure 1: Sample data from Guest Gale**

# Architecture overview

This project implements a containerized **ETL (Extract, Transform, Load)** pipeline to prepare data for analytics and recommendation services. The system is built using **Docker Compose** to orchestrate multiple services including Airflow, Spark, and Big -Query integration. Below is a breakdown of the architecture components and their roles in the pipeline.

A screenshot of a phone

AI-generated content may be incorrect.

< Figure > ! Update

|  |  |
| --- | --- |
| **Component** | **Version/Tag** |
| Apache Airflow | apache/airflow:2.10.5-python3.12 |
| Apache Spark | bitnami/spark:3.5.2 |
| JupyterLab | jupyter/pyspark-notebook:spark-3.5.0 |
| Docker Engine | Platform base |

**Pipeline Process**

**Data Source**

* **Format**: CSV files (e.g., TMDB Movie Dataset)
* **Role**: Acts as the raw dataset to be ingested.

**Data Ingestion**

* **Tool**: Apache Airflow
* **Containerized**: Yes (Docker)
* **Description**: Airflow DAGs orchestrate the loading of raw CSV datasets from external or local sources into the data pipeline. It automates scheduled ingestion and tracks lineage.

**Data Processing**

* **Tool**: Apache Spark (Bitnami image)
* **Containerized**: Yes (Docker)
* **Description**: Spark processes the ingested raw data by performing transformation, cleaning, and enrichment. The output is a cleaned dataset (CSV format) ready for warehousing.

**Data Warehouse**

* **Platform**: Google Big Query
* **Zones**:
  + **Staging Area / Landing Zone**: Temporarily holds transformed datasets before final loading.
  + **Cleaned Dataset**: Final structured dataset is loaded into Big Query for analysis and serving.

**Data Service**

* **Query Layer**: Big Query provides SQL-based analytics queries over the cleaned dataset.
* **Recommendation System Interface**: Cleaned data is used to develop ML-based recommendation systems that query processed features like genres, companies, and languages.

**Architecture Details** from Dockerfile, docker compose.yml

This section documents the containerized architecture used for the **ETL\_tmdb\_dataset** project. It utilizes Docker for environment isolation and service orchestration, consisting primarily of **Apache Airflow**, **Apache Spark**, and **JupyterLab (optional)** components. The configuration is defined in two core files: Dockerfile and docker-compose.yml.

**Dockerfile** (Custom Airflow Image)

**Base Image**

FROM apache/airflow:2.10.5-python3.12

* The base image is Airflow 2.10.5 with Python 3.12.

**System Dependencies**

USER root

RUN apt-get update && apt-get install -y curl unzip openjdk-17-jdk && apt-get clean

* Installs essential tools like curl, unzip, and Java 17 (required by Spark).

**Install Spark 3.5.2**

ENV SPARK\_VERSION=3.5.2 ...

RUN curl -L https://.../spark-${SPARK\_VERSION}.tgz | tar zx -C /opt

* Downloads and installs Spark 3.5.2.
* Spark is installed to /opt/spark

**Environment Variables**

ENV JAVA\_HOME=/usr/lib/jvm/java-17-openjdk-amd64

ENV SPARK\_HOME=/opt/spark

ENV PATH="${SPARK\_HOME}/bin:$PATH"

* Sets JAVA\_HOME and SPARK\_HOME.
* Adds Spark to system PATH.

**Install Python Dependencies**

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

* Installs Python packages needed by the project (likely includes pyspark, google-cloud, etc.).

**docker-compose.yml Analysis**

The docker-compose.yml file orchestrates the services. Here’s a breakdown of the main components:

**Service:**

**airflow-webserver, scheduler, triggerer, worker**

* Built using the custom Dockerfile.
* Connected to a shared volume (./dags:/opt/airflow/dags) and /opt/bitnami/spark.
* Uses spark://spark-master:7077 to submit jobs.
* Communicates via the airflow\_network.

**spark-master**

image: bitnami/spark:3.5.2

environment:

- SPARK\_MODE=master

* Runs Spark in master mode on port 7077 (for job submissions).

**spark-worker**

environment:

- SPARK\_MODE=worker

- SPARK\_MASTER\_URL=spark://spark-master:7077

* Connects to the Spark master to execute Spark jobs submitted via DAGs.

**Volumes and Mounts (Airflow & Spark)**

In this architecture, volumes are used to ensure that key files and outputs are shared across containers. This allows Airflow to trigger Spark jobs, Spark to process data, and both to access the same dataset and output.

|  |  |  |
| --- | --- | --- |
| **Host Path** | **Container Path** | **Purpose** |
| ./airflow/dags | /opt/airflow/dags | Contains Airflow DAG definitions |
| ./airflow/logs | /opt/airflow/logs | Stores Airflow task logs |
| ./airflow/data | /opt/airflow/data | Used for intermediate data or input/output specific to Airflow tasks |
| ./airflow/plugins | /opt/airflow/plugins | Custom Airflow plugins |
| ./airflow/config | /opt/airflow/config | Custom configuration files for Airflow |
| ./spark/app | /opt/bitnami/spark/app | PySpark application scripts (e.g., clean\_data.py, transform\_data.py) |
| ./spark/resources | /opt/bitnami/spark/resources | Raw dataset files used by Spark |
| ./spark/output | /opt/shared/output | Output directory for transformed parquet files shared with BigQuery loaders |

**Gantt Chart Plan (June–July 2025)**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Task** | **Week 1** | **Week 2** | **Week 3** | **Week 4** | **Week 5** | **Week 6** | **Week 7** | **Week 8** |
| **Phase 1: Design & Plan (June 2025)** |  |  |  |  |  |  |  |  |
| Learn Spark & BigQuery |  |  |  |  |  |  |  |  |
| Lab Implementation (Component Setup) |  |  |  |  |  |  |  |  |
| System Architecture Design |  |  |  |  |  |  |  |  |
| Phase 1 Report Writing |  |  |  |  |  |  |  |  |
| **Phase 2: Implementation (July 2025)** |  |  |  |  |  |  |  |  |
| Component Implementation |  |  |  |  |  |  |  |  |
| Integration Testing |  |  |  |  |  |  |  |  |
| System Demo Preparation |  |  |  |  |  |  |  |  |
| Final Presentation & Report |  |  |  |  |  |  |  |  |

**Data Processing**

From the *Architecture Overview* section, this project follows the ETL process. First, it **extracts data** from the raw dataset file. Second, it **transforms** the data by cleaning and reshaping it into a usable format. Finally, it **loads** the processed data into a data warehouse (Big Query).  
In this section, we describe the data processing steps in detail for this project.

**ETL\_tmdb\_dataset DAG**

The **ETL\_tmdb\_dataset** DAG is designed to automate the **end-to-end data pipeline** for the TMDB movie dataset. This pipeline implements the standard ETL process—**Extract**, **Transform**, and **Load**—to prepare and ingest data into **Big Query** for downstream analytics and machine learning applications.

The DAG is defined and executed within **Apache Airflow**, which provides orchestration, scheduling, and monitoring capabilities. It uses the following Airflow features:

**DAG Configuration Highlights**

* **DAG Definition**  
  The workflow is defined using Python in Airflow, giving full control over task structure, dependencies, and retry logic.
* **Manual Execution**  
  The DAG is set to run manually (schedule\_interval=None) and does not attempt to rerun missed intervals (catchup=False), making it ideal for on-demand or development workflows.
* **Task Groups**  
  The pipeline logic is organized into multiple **Task Groups**:
  + load\_to\_bigquery\_group for uploading data to BigQuery
  + validate\_bigquery\_group for post-upload validation  
    This modular structure improves readability and simplifies maintenance.
* **Operators Used**  
  The DAG leverages multiple Airflow operators:
  + PythonOperator for custom Python logic (e.g., file validation and BigQuery uploads)
  + SparkSubmitOperator to submit PySpark jobs to the Spark cluster
  + BigQueryTableExistenceSensor, BigQueryCheckOperator, and BigQueryGetDataOperator for validating data loads

**Connection Management**

The DAG uses Airflow's built-in **Connection IDs** to securely integrate with external services:

* **google\_cloud\_default**  
  Used to authenticate with **Google Cloud Platform (GCP)** via the GoogleBaseHook. It provides credentials for:
  + Uploading Parquet files to **BigQuery**
  + Performing validation tasks on BigQuery tables  
    This connection must be configured in the Airflow UI before DAG execution and typically uses a service account key with appropriate IAM roles.
* **spark\_default**  
  Used by SparkSubmitOperator to connect to the **Spark cluster** defined by the spark://spark-master:7077 URL.  
  This allows the DAG to offload heavy data processing tasks—like cleansing and transformation—to a distributed Spark runtime.  
  The connection can be configured with minimal setup, often just specifying the Spark master URL.

These connections enable **secure**, **centralized**, and **reusable** configuration without hardcoding credentials or cluster settings into the DAG.

< Figure Flow Diagram>

**DAG Tasks and Flow**

|  |  |
| --- | --- |
| **Task** | **Details** |
| Dataset Validation | **Task:** check\_dataset\_exists  **Description:** This task checks the existence of the input raw dataset file before proceeding to transformation steps. If the file is missing, the DAG fails early. |
| Data Cleansing (Spark) | **Task:** cleansing\_data  **Description:** Runs clean\_data.py using Spark to clean, filter, and normalize the raw TMDB data. This includes removing duplicates, validating schemas, and preparing the data for transformation. |
| Data Transformation (Spark) | **Task**: transform\_data  **Description**: Executes transform\_data.py using Spark to create structured dimensional and fact tables (e.g., dim\_movie, fact\_movie) along with bridge tables (e.g., bridge\_movie\_genre) in parquet format. |
| Load to Big Query | **Task Group**: load\_to\_bigquery\_group  **Description**: Iteratively uploads the output parquet files into their corresponding BigQuery tables. It is divided into:   * Dimension tables: dim\_movie, dim\_genre, etc. * Bridge tables: bridge\_movie\_genre, etc. * Fact table: fact\_movie |
| Validate Big Query Tables | **Task Group**: validate\_bigquery\_group  **Description**: For each table loaded into BigQuery:   * Checks if the table exists. * Verifies that the table has at least one row. * Optionally samples the schema by retrieving the first row |

**Data Cleansing**

The **data cleansing** process is a critical part of the ETL pipeline, handled by the **cleansing\_data** task in the DAG. This task executes the **clean\_data.py** script using Apache Spark and focuses on cleaning, validating, and standardizing the raw movie data before any transformation or loading steps occur.

Key operations performed during this phase include:

1. **Schema Validation**  
   The script validates that the input dataset conforms to an expected schema. It ensures the presence and correct order of critical columns such as movie\_id, title, genres, keywords, and overview. Any deviations result in an early failure to maintain data quality.
2. **Data Type Enforcement**  
   The data types of important columns are checked:
   * movie\_id must be an integer.
   * Columns like title, genres, keywords, and overview must be strings.  
     This step ensures compatibility with downstream Spark operations and schema consistency.
3. **Duplicate Removal**  
   The script removes duplicate entries based on the **movie\_id** column. Only the last occurrence is retained using Spark’s **dropDuplicates()** method to preserve the most up-to-date or complete entry.
4. **Missing Value Filtering**  
   Records with null or invalid values in essential fields are filtered out. This reduces the risk of errors in later processing stages and improves the quality of data ingested into Big Query.
5. **Date Filtering**  
   Movies with invalid or future **release\_date** values are excluded. The script filters out dates that:
   * Are beyond a cutoff point (e.g., **release\_date** > 2025-06-02)
   * Precede a historically valid date (e.g., before 1582-10-15)
6. **NA and Placeholder Cleanup**  
   Non-informative placeholders (like 'NA') in numeric fields such as budget, runtime, and revenue are replaced with zero. This avoids computation errors during analytics.
7. **Released-Only Filtering**  
   Only movies with the status field marked as "Released" are retained. This ensures that unreleased or canceled projects are excluded from the final dataset.
8. **String Cleanup**  
   Double quotes and unnecessary characters are stripped from columns like title, overview, and keywords to improve formatting and text vectorization downstream.
9. **Country Name Normalization**  
   The **production\_countries** column is cleaned by splitting comma-separated values, lowercasing and trimming country names, and joining with a reference ISO country list to ensure standardized country codes.

After cleansing, the dataset is well-structured and standardized, with irrelevant, invalid, or malformed entries removed. The resulting DataFrame is passed into the transformation step, where it is further reshaped into dimensional and fact models.

**Data Transformation** (Create Dimension Table and Fact Table)

Following the data cleansing step, the **data transformation** process is executed through the transform\_data task in the DAG. This step utilizes Apache Spark to reshape the cleaned dataset into a structured format suitable for loading into a star schema-based data warehouse.

The script **transform\_data.py** performs the following key operations:

**1. Dimensional Table Creation**

The cleaned dataset is used to generate several dimension tables. These tables store descriptive attributes and enable slicing and filtering of fact data.

* **dim\_movie**: Contains core movie metadata such as title, release date, original language, and overview.
* **dim\_keyword**: Extracted from the keywords field, which is parsed and exploded into individual rows.
* **dim\_genre**: Created from the structured genre list extracted during cleansing.
* **dim\_production\_company**, **dim\_production\_country**, **dim\_spoken\_language**: Each table is normalized from the respective columns using explode and join operations with reference data when available.

Each dimension table has a unique identifier and contains only distinct entries.

**2. Bridge Table Creation**

Because many of the relationships between movies and their attributes are **many-to-many** (e.g., a movie can have multiple genres or companies), **bridge tables** are created to model these relationships.

* **bridge\_movie\_genre**
* **bridge\_movie\_keyword**
* **bridge\_movie\_company**
* **bridge\_movie\_country**
* **bridge\_movie\_language**

Each bridge table includes **movie\_id** and the corresponding foreign key from its related dimension. These tables are important for maintaining relational integrity when querying in BigQuery.

**3. Fact Table Creation**

The fact table fact\_movie is built to hold quantitative metrics and foreign keys for analytical queries.

* Includes measures such as vote\_count, vote\_average, popularity, budget, revenue, and runtime.
* Also includes the movie\_id, which links to all bridge and dimension tables.

This table enables numerical aggregation and joins with dimensional attributes for analytics.

**4. Save as Parquet**

Each output table (dimension, bridge, fact) is saved in **Parquet format** to the /opt/shared/output directory. Parquet offers high performance for columnar storage, making it efficient for both reading and uploading into Big Query.

The transformation process produces a star-schema-compatible dataset with clean, normalized, and well-structured data. These Parquet outputs serve as the input for the next phase: loading into Big Query.

**Load to Data Warehouse (Big Query)**

Once the data has been transformed and saved into Parquet format, the final phase of the pipeline is to load it into Big Query, Google Cloud's serverless data warehouse. This step is orchestrated within the DAG using the **load\_to\_bigquery\_group** task group, which handles the upload of each table individually, followed by post-load validation using the **validate\_bigquery\_group**.

**Upload Process**

**File Detection**  
All .parquet part files in each output directory are discovered and uploaded sequentially.

**Write Mode Handling**

* The **first file** is loaded using WRITE\_TRUNCATE, which overwrites any existing data in the table.
* **Subsequent files** use WRITE\_APPEND to append records without overwriting.

**Parquet Optimization**  
Since Spark outputs partitioned data in multiple Parquet files, this loading method ensures all parts are included efficiently.

**Credentials and Connection Management**  
The function uses Airflow’s built-in **connection ID** google\_cloud\_default, which you configure through the Airflow UI prior to running the DAG.  
This connection securely provides credentials via GoogleBaseHook, ensuring that:

* No hardcoded secrets are stored in the code.
* Credentials are scoped and managed via Airflow’s centralized system.
* The connection supports role-based access control and auditability.

**Post-Load Validation**

To ensure data integrity and availability, each table undergoes a three-step validation process defined in the validate\_bigquery\_group:

1. **Table Existence Check**  
   A **BigQueryTableExistenceSensor** confirms whether the target table exists in the dataset after loading.
2. **Row Count Verification**  
   A **BigQueryCheckOperator** runs a query to ensure the table contains at least one record, confirming successful ingestion.
3. **Schema Sampling (Optional)**  
   A **BigQueryGetDataOperator** fetches the first row from the table to inspect its schema and confirm structural validity.

These checks run in sequence to guarantee that the loaded tables are fully functional, well-formed, and populated.

This step successfully moves the transformed dataset into the cloud-based data warehouse (BigQuery), ready for further analytics, reporting, or machine learning tasks. The combination of **automated loading**, **secure authentication via Airflow Connections**, and **systematic validation** ensures reliability and scalability of the data pipeline.

====

**Export\_Recom\_Dataset\_from\_GCP DAG**

The **Export\_Recom\_Dataset\_from\_GCP** DAG is designed to automate the data export process for the content-based movie recommendation dataset generated and stored in Big Query. This DAG handles exporting, consolidating, downloading, and validating the final recommendation dataset to prepare it for external use—such as model serving or analytical visualization.

This workflow complements the **ETL\_tmdb\_dataset** DAG by operating after the transformed dataset has already been loaded into Big Query and structured as a view (**cbf\_movie\_recommendations\_view**).

**DAG Configuration**

* **Trigger Mode: Manual (schedule\_interval=None)**
* **Catchup: Disabled (catchup=False)**
* **Owner: Polakorn Anantapakorn**
* **Technology Stack:**
  + **Google BigQuery: Export source**
  + **Google Cloud Storage (GCS): Intermediate and final storage**
  + **Airflow Local Filesystem: Final validation target**

**DAG Tasks and Flow**

|  |  |  |
| --- | --- | --- |
| **Task** | **Operator** | **Description** |
| export\_view\_to\_gcs | BigQueryInsertJobOperator | Exports data from the BigQuery view cbf\_movie\_recommendations\_view into multiple CSV shard files stored in a GCS bucket (gs://tmdb-reco-flow-bucket/output/) |
| merge\_csv\_shards | PythonOperator | Executes the custom Python function merge\_gcs\_csv\_shards to combine multiple CSV shards into a single file (final/cbf\_movie.csv) in the same GCS bucket |
| download\_from\_gcs | GCSToLocalFilesystemOperator | Downloads the merged CSV file from GCS to the local Airflow directory at /opt/airflow/data/cbf\_movie.csv |
| validate\_data | PythonOperator | Runs the validate\_csv function to validate schema, data types, and null constraints in the downloaded file before downstream consumption |

All tasks are defined using Airflow-native operators and chained sequentially:

**export\_bq >> merge\_csv >> download >> validate**

**Connection Management**

The DAG uses the predefined Airflow connection ID:

* **google\_cloud\_default**: Authenticates with Google Cloud Platform for both **Big Query** and GCS operations. This connection should be preconfigured in the Airflow UI and scoped with the appropriate service account credentials for:
  + Running **Big Query** export jobs
  + Accessing GCS files for read/write
  + Downloading data securely to the Airflow instance

**Data Validation**

The final task in the DAG, validate\_data, runs a custom Python function validate\_csv to ensure the exported CSV file is complete, clean, and trustworthy before it is used by downstream systems.

This function performs several **quality checks** on the downloaded file (/opt/airflow/data/cbf\_movie.csv) using the pandas library. The validation includes the following steps:

1. **Column Name Validation**  
   The function checks whether the columns in the CSV exactly match the expected schema:  
   ["movie\_id", "title", "genres", "keywords", "overview"].  
   Any schema mismatch will trigger an exception.
2. **Data Type Validation**
   * movie\_id must be of type **integer**.
   * All text-based fields (title, genres, keywords, overview) must be of **string** type.
3. **Null Value Check**  
   The function verifies that there are no missing values in any field. If any nulls are found, a detailed report is logged and the DAG fails.
4. **Duplicate ID Check**  
   It ensures that all movie\_id values are **unique**. Duplicate rows are flagged and shown in the error log.
5. **Text Length Check**  
   Empty or whitespace-only values are not allowed in title and overview. The function scans for and flags such records to prevent low-quality descriptions from entering production.
6. **Corrupt Character Detection**  
   The function checks for the presence of the Unicode replacement character �, which may indicate encoding issues or corrupt text data. Any field containing this character will fail the validation.

All errors encountered during validation are logged, and an exception is raised to halt the pipeline, ensuring that only high-quality, production-ready data is allowed to pass through.

**Outcome**

By integrating this validation step, the DAG guarantees that the exported recommendation dataset adheres to strict data quality standards before being shared or served. This helps prevent model degradation, incorrect recommendations, and downstream data integrity issues.

**Result**

The **Export\_Recom\_Dataset\_from\_GCP** DAG successfully delivers a single, validated CSV file representing the output of a content-based recommendation system. It provides a clean, ready-to-use dataset for use cases such as:

* Deployment to machine learning APIs
* External reporting tools (e.g., Data Studio, Tableau)
* Batch delivery to data consumers or partners

By automating the export and validation steps, this DAG ensures consistency, correctness, and reproducibility of exported recommendation data.

====