A Steam Game Recommendation System

IE 7374: Data Pipeline Assignment

Team Members:

Akshun Singh
Balasubramaniam Renganathan
Moheesh Kavitha Arumugam
Naga Venkata Sai Sreeya Galla
Pooja Laxmi
Sankarakameswaran
Sruthi Gandla

Introduction

Business Problem:

Steam Select is a machine learning-powered game recommendation system designed to enhance user experience by providing personalized game suggestions and profitable bundle deals. Users input five of their favorite games, and the system recommends five similar games along with profitable game bundles, helping them discover new titles efficiently.

Dataset:

The dataset originates from Julian McAuley's repository but has been sampled to 25% of the original user reviews to accommodate storage and compute constraints. The dataset covers over 16,000 games with the following tables:

- Bundle Data: 615 rows
- Item Metadata: 32,100 rows
- Reviews: 1,100,000 rows

The dataset can be downloaded from Hugging Face.

Manual Download Instructions:

To manually download the dataset from Hugging Face:

- Visit the dataset page: *<https://huggingface.co/datasets/PookiePooks/steam-games-dataset/tree/main>*
- Click on the "Download" button.
- Extract the dataset files and place them in the GCS bucket.

Installation

This project requires Python >= 3.9. Additionally, this project is compatible with Windows, Linux,

and Mac operating systems.

Prerequisites

- git
- python>=3.9
- docker desktop should be running

User Installation

The User Installation Steps to follow:

1. Clone the git repository onto your local machine:
   git clone https://github.com/your-username/steam-select.git
   cd steam-select

2. Check if python version >= 3.9 using this command:
   python --version

3. Check if you have enough memory
   *docker run --rm "debian:bullseye-slim" bash -c 'numfmt --to iec $(echo $(($(getconf _PHYS_PAGES) * $(getconf PAGE_SIZE))))'*

   NOTE: If you get the following error, please increase the allocation memory for docker.

   Error: Task exited with return code -9 or zombie job

4. After cloning the git onto your local directory, please edit the docker-compose.yaml with the following changes:
   AIRFLOW__SMTP__SMTP_HOST: *smtp.gmail.com # If you are using other than gmail to send/receive alerts change this according to the email provider.*
   AIRFLOW__SMTP__SMTP_USER: *# Enter your email 'don't put in quotes'*
   AIRFLOW__SMTP__SMTP_PASSWORD: *# Enter your password here generated from google in app password*
   AIRFLOW__SMTP__SMTP_MAIL_FROM: *# Enter your email*
   *- **${AIRFLOW_PROJ_DIR:-.}/dags:** #locate your dags folder path here (eg: C:\GitHub\Steam-Select\dags)*
   *- **${AIRFLOW_PROJ_DIR:-.}/logs:** #locate your project working directory folder path here (eg: C:\GitHub\Steam-Select\logs)*
   *- **${AIRFLOW_PROJ_DIR:-.}/config:** #locate the config file from airflow (eg: C:\GitHub\Steam-Select\config)*
5. In the cloned directory, navigate to the config directory under **Steam-Select** and place your key.json file from the GCP service account for handling pulling the data from GCP.

6. Build the Docker Image
   docker compose build

7. Run the Docker composer and initialize airflow.
   docker compose up ariflow-init

8. Run the docker image.
   docker compose up

9. To view Airflow dags on the web server, visit https://localhost:8080 and log in with credentials
   user: airflow
   password: airflow

10. Run the DAG by clicking on the play button on the right side of the window

11. Stop docker containers (run *docker compose down* in the terminal)

Tools Used for MLOps

- GitHub Actions
- Docker
- Airflow
- Data Version Control (DVC)
- Google Cloud Storage (GCS)

GitHub Actions

- GitHub Actions is configured to initiate workflows upon pushes and pull requests to any branch, including the "feature" and main branches.
- When a new commit is pushed, the workflow triggers a build process and unit tests. This process generates test reports in XML format, which are stored as artifacts. The workflow is designed to locate and execute test cases situated within the test directory that correspond to modules in the DAGs directory.
- Additionally, the workflow assesses the code for readability, potential security issues, and adequate documentation. Upon the successful completion of these build checks, feature branches are merged into the main branch.

Docker and Airflow

The docker-compose.yaml file contains the code necessary to run Airflow. Through the use of Docker and containerization, we are able to ship our data pipeline with the required dependencies installed. This makes it platform independent, whether it is windows, mac or linux, our data pipeline should run smoothly.

Data Version Control (DVC)

DVC (Data Version Control) is an open-source tool essential for **data versioning** in machine learning projects. It tracks changes in datasets over time, ensuring **reproducibility and traceability** of experiments. By storing meta-information separately from data, DVC keeps Git repositories **clean and lightweight**.

DVC integrates seamlessly with Git, allowing for efficient management of **code, data, and models**. This **dual-repository** approach simplifies collaboration and ensures that project states can be recreated easily. Given that our project deals with **static game data**, we use DVC to manage our dataset versions while ensuring data consistency throughout the pipeline.

DVC's focus on **data versioning** is critical for maintaining the **integrity and reliability** of our machine learning workflows.

Google Cloud Platform (GCP)

We utilized Google Cloud Storage exclusively for storing our raw data and processed data ensuring they are securely archived and readily accessible.

Steps to set up a service account to use Google Cloud Platform services are as follows:

1. Go to the GCP Console: https://console.cloud.google.com/.
2. Navigate to **IAM & Admin > Service accounts**: In the left-hand menu, click on "IAM & Admin" and then select "Service accounts".
3. **Create a service account:** Click on the "Create Service Account" button and follow the prompts. Give your service account a name and description.
4. **Assign permissions:** Assign the necessary permissions to your service account based on your requirements. You can either grant predefined roles or create custom roles.
5. **Generate a key:** After creating the service account, click on it from the list of service accounts. Then, navigate to the "Keys" tab. Click on the "Add key" dropdown and select "Create new key". Choose the key type (JSON is recommended) and click "Create". This will download the key file to your local machine.

You can download our dataset and follow these steps to create a GCP bucket and upload the files in the bucket add your GCP account's key.json.

Schema and Statistics Generation

- Schema generation is typically used in structured data pipelines where schema enforcement ensures data consistency across transformations. However, in our pipeline:

- The dataset structure is inherently stable, sourced from a well-defined public repository with consistent formats (CSV/JSON/Parquet).

- Any required schema validation (column types, missing values, etc.) is already handled within our data validation pipeline through explicit checks.

- Since our feature engineering is based on dynamic user inputs and real-time inference, enforcing a rigid schema could introduce unnecessary constraints rather than improving pipeline robustness.

Thus, schema generation does not provide additional value to our workflow.

Bias Detection and Mitigation

- Bias detection and mitigation are critical in areas such as hiring, credit scoring, or medical applications, where fairness across demographic groups is essential. However, for our project:

- No sensitive demographic attributes (age, gender, ethnicity, etc.) are present in the dataset, making traditional bias detection methods irrelevant.

- Our recommendations are based on user preferences and game metadata (genre, reviews, sentiment scores), rather than user-specific attributes that could introduce societal biases.

- The primary goal is to suggest games similar to user preferences, and our model does not involve decisions that adversely impact any particular group.

Given the absence of sensitive attributes and the nature of our application, bias detection and mitigation are not required for our recommendation system.

Unit Testing and Failure Alerts in Preprocessing

- Unit testing has been fully implemented across all preprocessing steps, ensuring that each transformation is validated before proceeding. Specifically:

- Each individual preprocessing script (e.g., clean_reviews.py, clean_item.py, clean_bundle.py) includes inline unit tests to verify data integrity at key transformation stages.

- Failure alerts have been embedded within these scripts to immediately flag errors during data preprocessing. In case of failure:

- Log messages capture the exact issue encountered.

- Airflow failure alerts notify stakeholders when a DAG step fails.

- Graceful failure handling ensures that incomplete or erroneous data does not propagate downstream.

By incorporating unit tests within each script and setting up a robust alerting system, we ensure high reliability and early detection of potential data inconsistencies in the pipeline.

Pipeline Optimization

We created 4 Data Pipelines:

1. **Data Download**

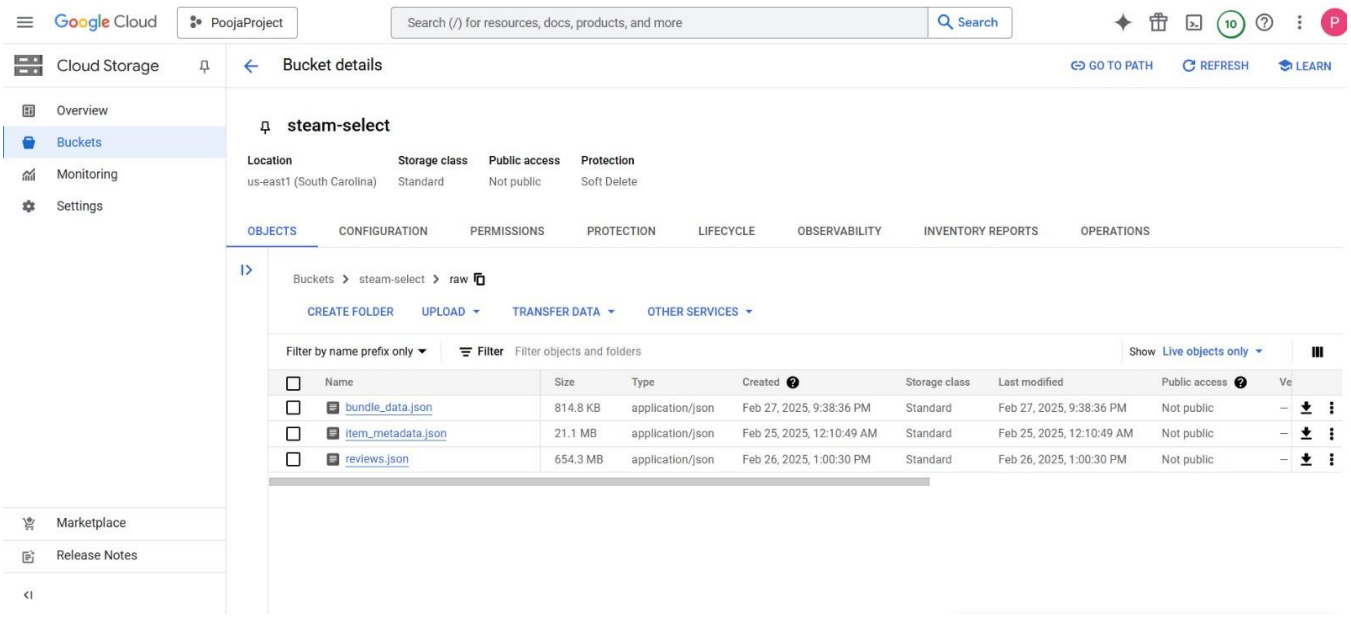
2. **Data Cleaning**
3. **Data Validation**
4. **Feature Engineering**

Our Pipeline begins with data acquisition where we fetch Data from the Source. After the Data is downloaded, in the next step, we clean and Preprocess the Data. Next, the preprocessed data is validated and feature engineered.

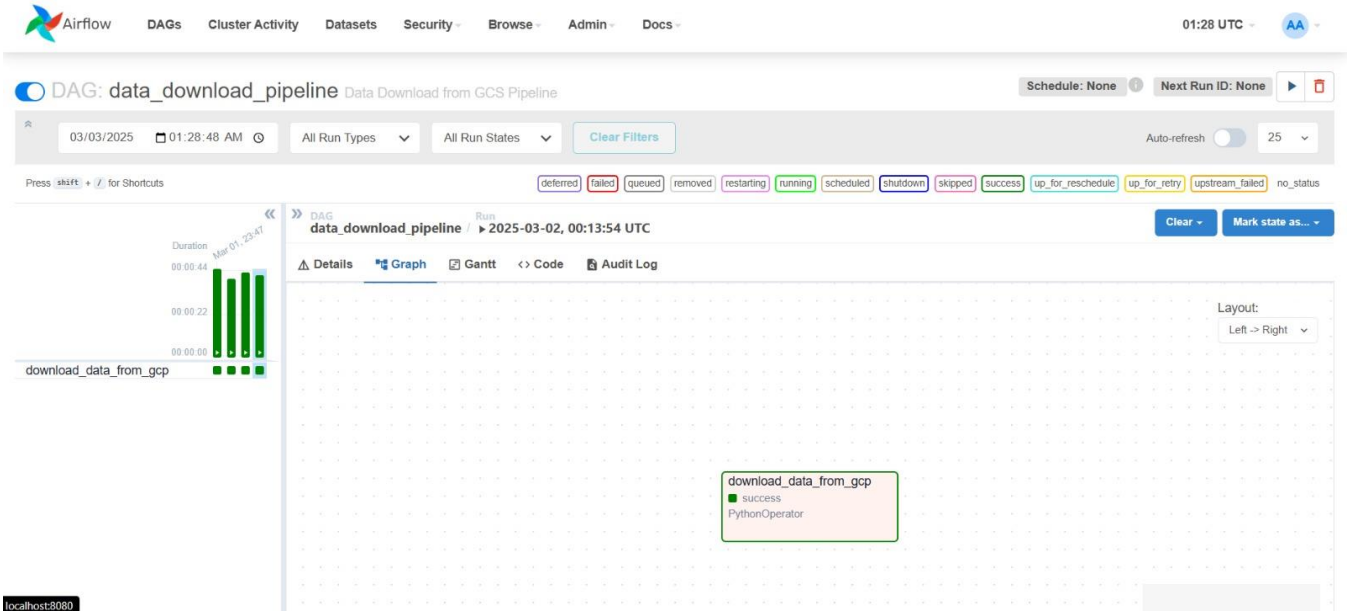
We use Apache Airflow to orchestrate our data pipeline, treating each module as a distinct task within our primary DAG (Directed Acyclic Graph). This setup allows us to efficiently manage the flow from data acquisition to model deployment, ensuring each step is executed in the correct order and monitored for performance and success.

Images of Excecution of our Dags:

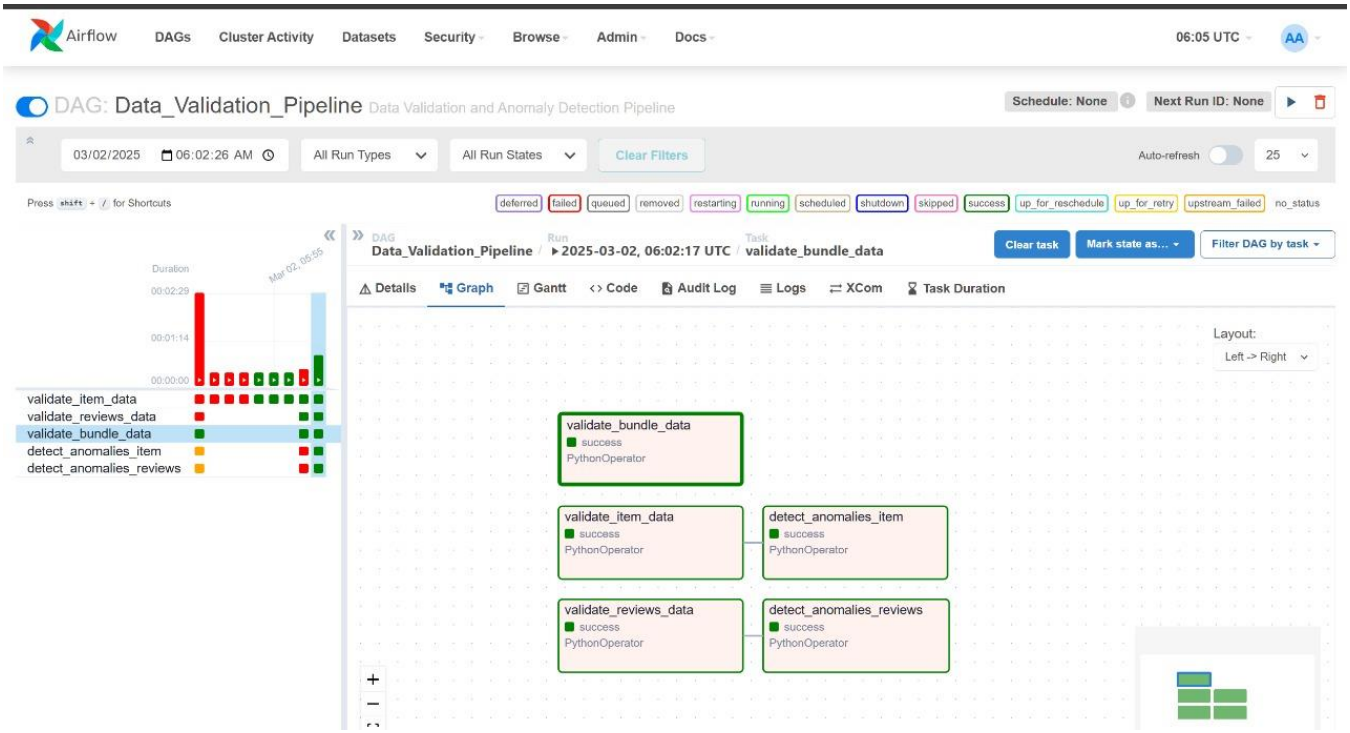
Data Download Pipeline Input:



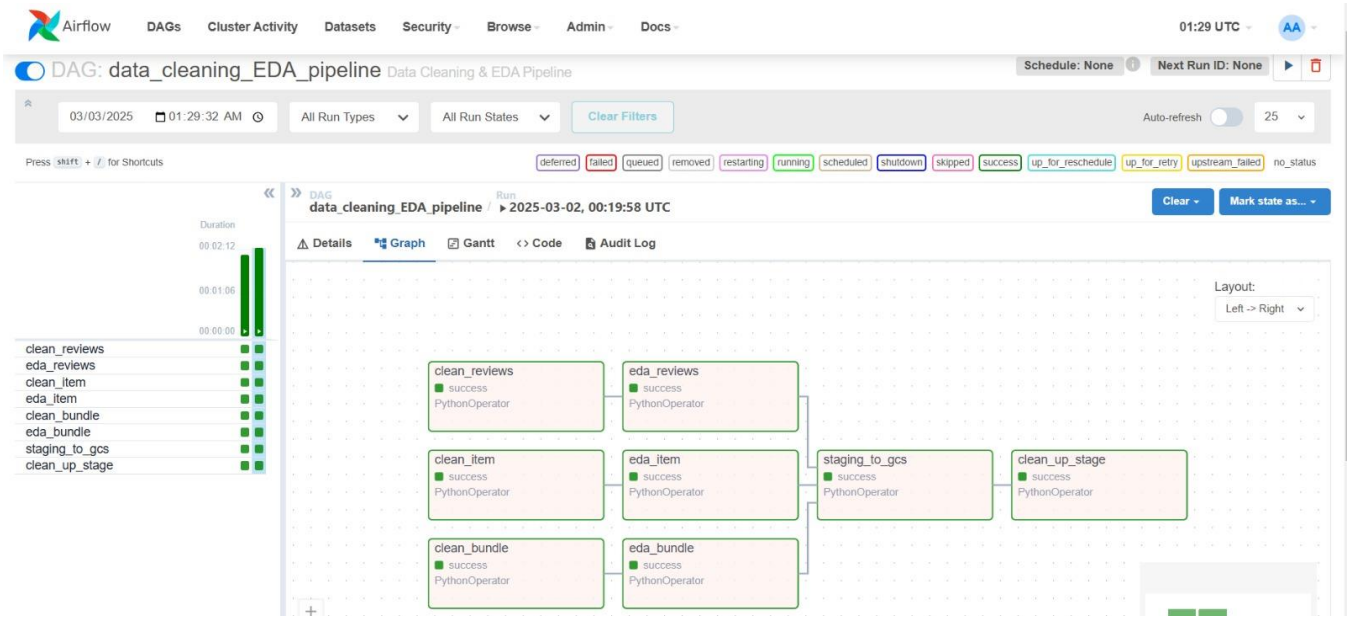
Data Download Pipeline Results:



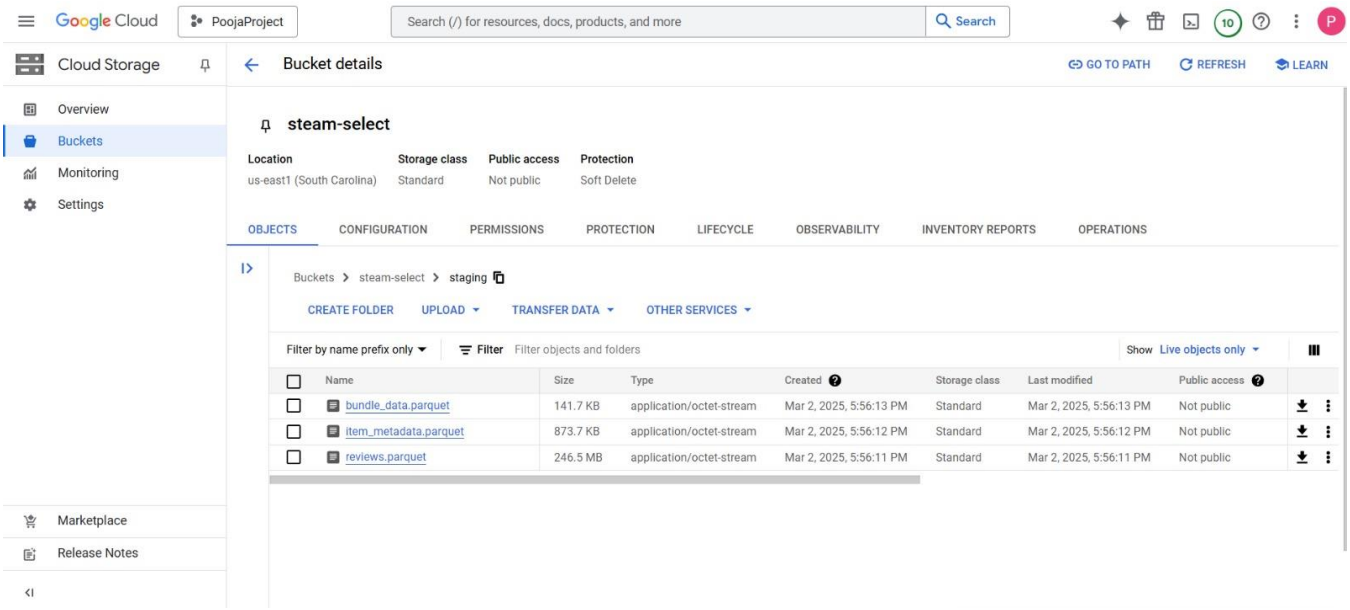
Data Validation Pipeline Results:



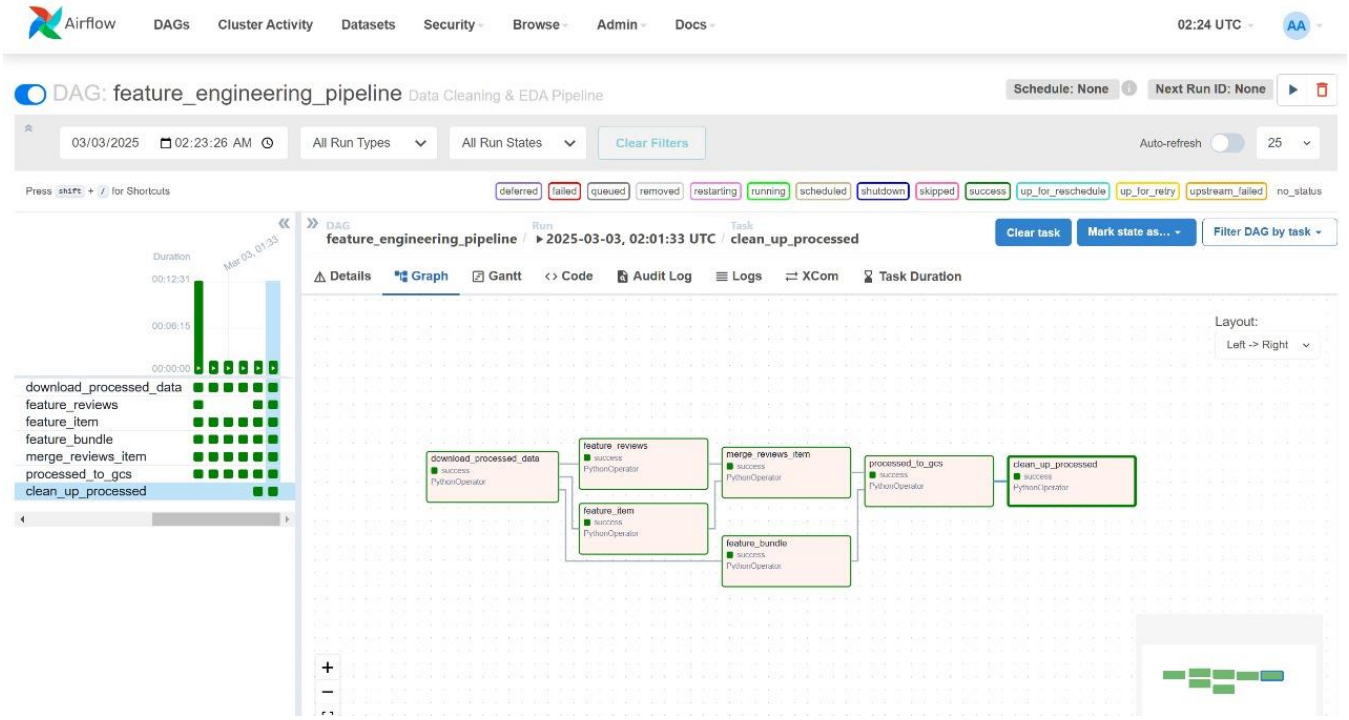
Data Cleaning & EDA Pipeline Results:



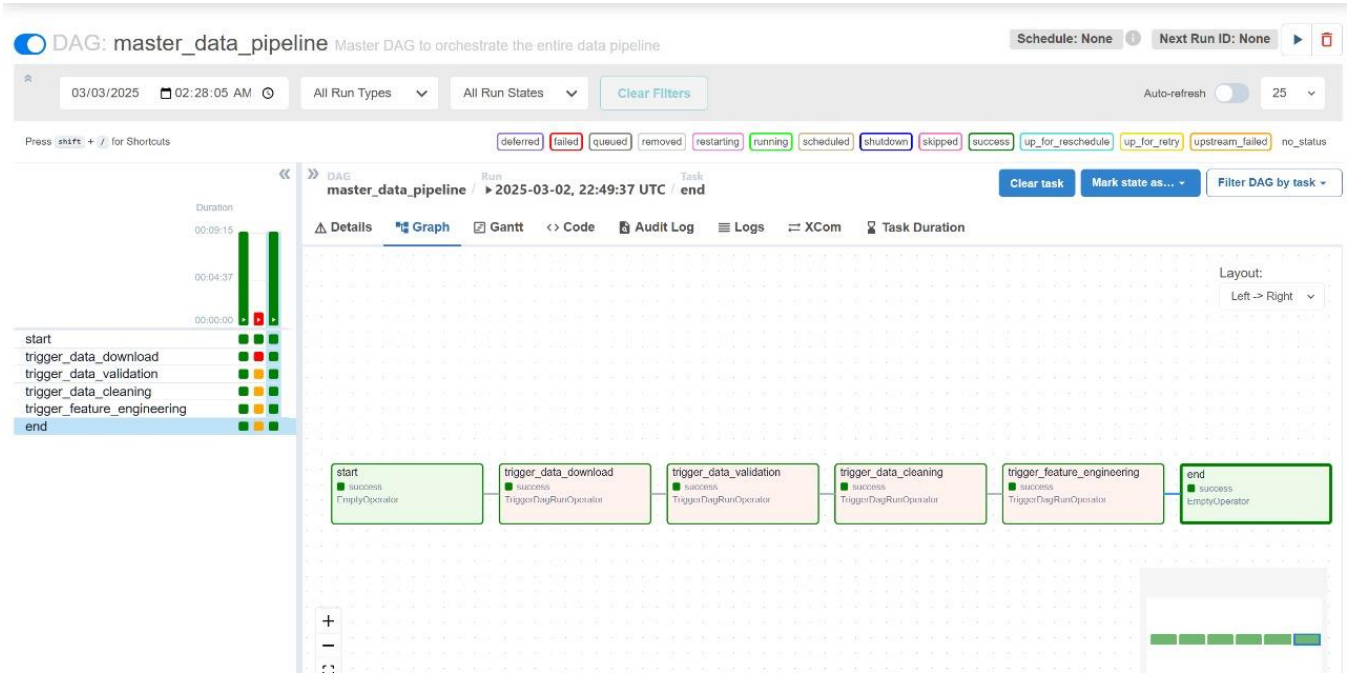
Data Cleaning & EDA Pipeline Output Stored in GCS:



Feature Engineering Pipeline Results:



Master Data Pipeline results:



Data Download

In this phase we downloaded the data from GCS and stored into the designated folders using the following module:

download_data.py: The python file downloads the data from GCP bucket and stores it into the in the designated folder under the project directory.

Data validation

In this phase, the dataset undergoes a series of validation checks to ensure data integrity, consistency, and quality before further processing. The validation pipeline consists of the following modules:

Data Validation Checks

- validate_data.py: This script performs essential validation checks on the dataset, ensuring data integrity before it progresses in the pipeline. It:
- Identifies and logs missing values in the dataset.
- Detects and removes duplicate records to maintain data consistency.
- Ensures numerical columns do not contain negative values where they are not expected.
- Converts mixed-type columns (such as price) into numeric format to handle inconsistencies.
- Saves validation results in a validation_report.txt file within the Data Validation Reports directory.
- Logs any detected issues into a dedicated validation.log file.
- **read_and_validate_file()**: This function reads JSON data from the raw dataset folder, validates it using the validate_data function, and logs any inconsistencies or data integrity issues.

Anomaly detection and alerts

- anomaly_detection.py: This script identifies anomalies in the dataset based on predefined rules for data quality and structure, logging any detected issues. It:
- Checks numerical columns for unexpected negative values.
- Detects unexpected categorical values that do not belong to predefined sets (e.g., marital status, education level).
- Saves anomaly detection results in a anomaly_results.json file within the Data Validation Reports directory.
- Logs any anomalies found in a anomaly.log file.
- If anomalies are detected, the script triggers an alert mechanism.

- read_and_detect_anomalies(): This function loads JSON data from the raw dataset folder, detects anomalies using predefined rules, and logs the detected issues for further review.

Data Validation Pipeline (Airflow Integration)

- The Data Validation Pipeline is managed using Apache Airflow, ensuring automated execution of validation and anomaly detection tasks in sequence.
- data_validation_pipeline.py: This Airflow DAG (Directed Acyclic Graph) schedules and executes data validation and anomaly detection as separate tasks. The pipeline:
- Reads and validates datasets (item_metadata.json, reviews.json, bundle_data.json) using validate_data.py.
- Detects anomalies in datasets using anomaly_detection.py.
- Logs all validation and anomaly detection results in the respective log files.
- Defines dependencies, ensuring that validation occurs before anomaly detection.

Data Cleaning and Preprocessing

The data cleaning pipeline consists of multiple scripts that process different datasets:

1. Cleaning Bundle Data

- **Script**: clean_bundle.py
- **Key Tasks**:
  - Removes duplicate records.
  - Converts columns (bundle_final_price, bundle_price, bundle_discount) into appropriate numerical formats.
  - Parses items column from a string representation of lists into actual lists.
  - Standardizes column names.
  - Saves the cleaned data as a **Parquet** file for efficient storage and retrieval.

2. Cleaning Item Metadata

- **Script**: clean_item.py
- **Key Tasks**:
  - Drops irrelevant columns such as tags, reviews_url, publisher, metascore, and developer.
  - Ensures numerical columns (id) are properly cast to integer.
  - Maps sentiment labels to numerical scores for improved downstream analysis.
  - Standardizes missing values and renames columns (id → Game_ID, app_name → Game).
  - Saves the cleaned data as a **Parquet** file.

3. Cleaning Reviews Data

- **Script**: clean_reviews.py
- **Key Tasks**:
    - Removes duplicate records and trims whitespace.
    - Converts timestamps into readable date format.
    - Drops non-essential fields like compensation, user_id, username, etc.
    - Ensures numerical fields (product_id) are properly formatted.
    - Saves the cleaned data as a **Parquet** file.

4. Cleanup Staging

- **Script**: cleanup_stage.py
- **Key Tasks**:
    - Removes all temporary files and directories within the **processed data** folder.
    - Ensures that the system does not accumulate unnecessary files, keeping storage
    - Reliable and high-quality data is crucial for machine learning model performance. This preprocessing pipeline uses several scripts to ensure the data integrity and readiness for model training, making the pipeline more efficient and robust.

- **datatype_format.py:** Ensures data types are consistent across the dataset, improving compatibility with various analysis and machine learning techniques. It loads the processed data, checks and converts data types, logs data type changes

Exploratory Data Analysis (EDA)

EDA scripts analyze data distributions, missing values, and feature relationships through visualizations.

1. Bundle Data Analysis

- **Script**: EDA_bundle.py
- **Key Tasks**:
    - Creates distribution plots of bundle prices.
    - Analyzes bundle discounts against prices.
    - Generates box plots to detect outliers.

2. Item Metadata Analysis

- **Script**: EDA_item.py
- **Key Tasks**:
    - Visualizes sentiment scores.
    - Identifies top-rated games based on sentiment.
    - Analyzes genre distribution.

3. Reviews Data Analysis

- **Script**: EDA_reviews.py
- **Key Tasks**:
    - Analyzes the number of reviews per product.

- Generates histograms of review counts.

Data Processing Pipeline (Airflow Integration)

The **Data Cleaning and Preprocessing Pipeline** is orchestrated using **Apache Airflow**, ensuring an automated workflow for data transformation.

- **Script**: data_cleaning_pipeline.py
- **Key Steps**:
  - **Downloads raw data** from Google Cloud Storage (download_data.py).
  - **Performs data cleaning** using individual cleaning scripts.
  - **Runs EDA** tasks to generate insights.
  - **Uploads cleaned datasets** to Google Cloud Storage (write_to_gcs.py).
  - **Removes staging files** after processing to maintain storage efficiency

Feature Engineering Pipeline

The feature engineering pipeline consists of multiple scripts that enhance and refine the cleaned data:

1. Feature Engineering for Reviews

- **Script**: feature_reviews.py
- **Key Tasks**:
  - Analyzes text-based reviews to extract sentiment-based features.
  - Computes additional features like review length, number of positive/negative words, and review timestamps.
  - Merges extracted features into the cleaned reviews dataset.
  - Saves the feature-engineered dataset as a **Parquet** file for further processing.

2. Feature Engineering for Item Metadata

- **Script**: feature_item.py
- **Key Tasks**:
  - Extracts and standardizes categorical features such as game genre and publisher.
  - Computes new numerical attributes like average rating, weighted sentiment scores, and popularity ranking.
  - Encodes categorical variables to facilitate machine learning applications.
  - Saves the transformed dataset as a **Parquet** file.

3. Feature Engineering for Bundle Data

- **Script**: feature_bundle.py
- **Key Tasks**:
  - Extracts pricing features such as price per item and discount percentage.
  - Calculates total bundle value and savings percentage.
  - Generates metadata fields that assist in analyzing bundle performance.

- Saves the transformed dataset as a **Parquet** file.

4. Merging Reviews with Item Metadata

- **Script**: merge_reviews_item.py
- **Key Tasks**:
  - Merges the processed **reviews** and **item metadata** datasets.
  - Aligns sentiment scores with game IDs to facilitate sentiment-based analytics.
  - Ensures all missing values are handled before merging.
  - Outputs a merged dataset for modeling and analytics.

Notification

Email Notification Setup : The DAG in this project is configured to send email notifications upon task completion (both success and failure), ensuring that users stay informed about the pipeline's execution status without manually checking the Airflow dashboard. This notification system enhances monitoring and enables timely responses to task failures.

The notification system is implemented using the EmailOperator from Apache Airflow. The notification.py script contains functions that trigger emails when a task either succeeds or fails.

Functions in notification.py

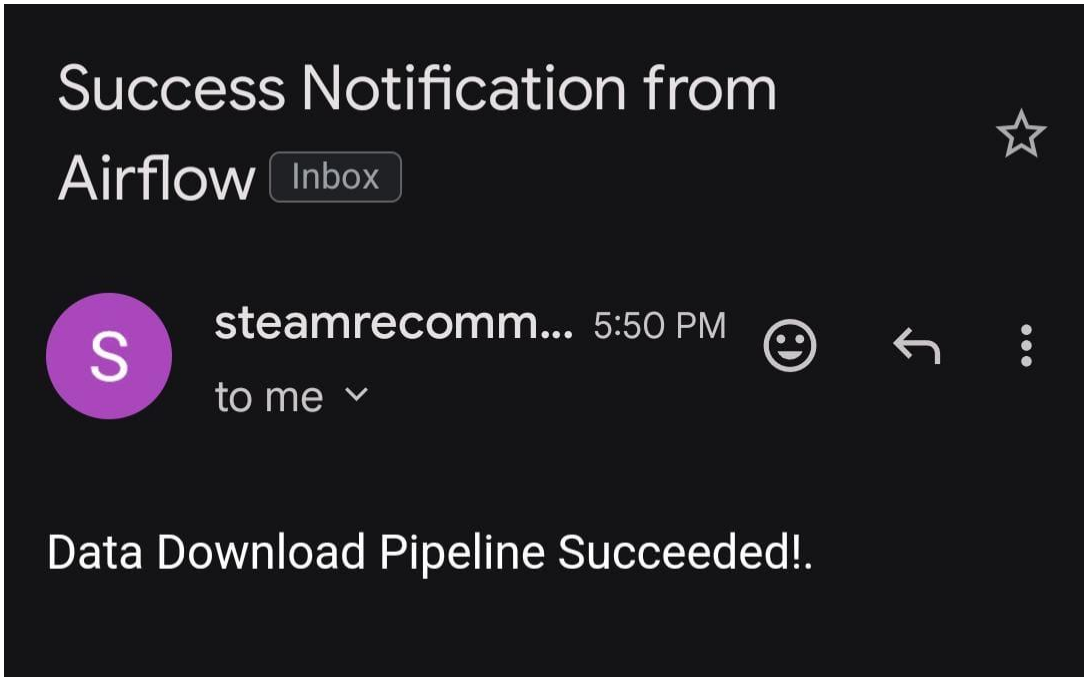
1. **notify_success(context, message)**:
   - Sends an email upon task success.
   - The email includes the subject **"Success Notification from Airflow"**.
   - The message body contains a success confirmation.
2. **notify_failure(context, message)**:
   - Sends an email when a task fails.
   - The email includes the subject **"Failure Notification from Airflow"**.
   - The message body details the failure, allowing users to investigate issues promptly

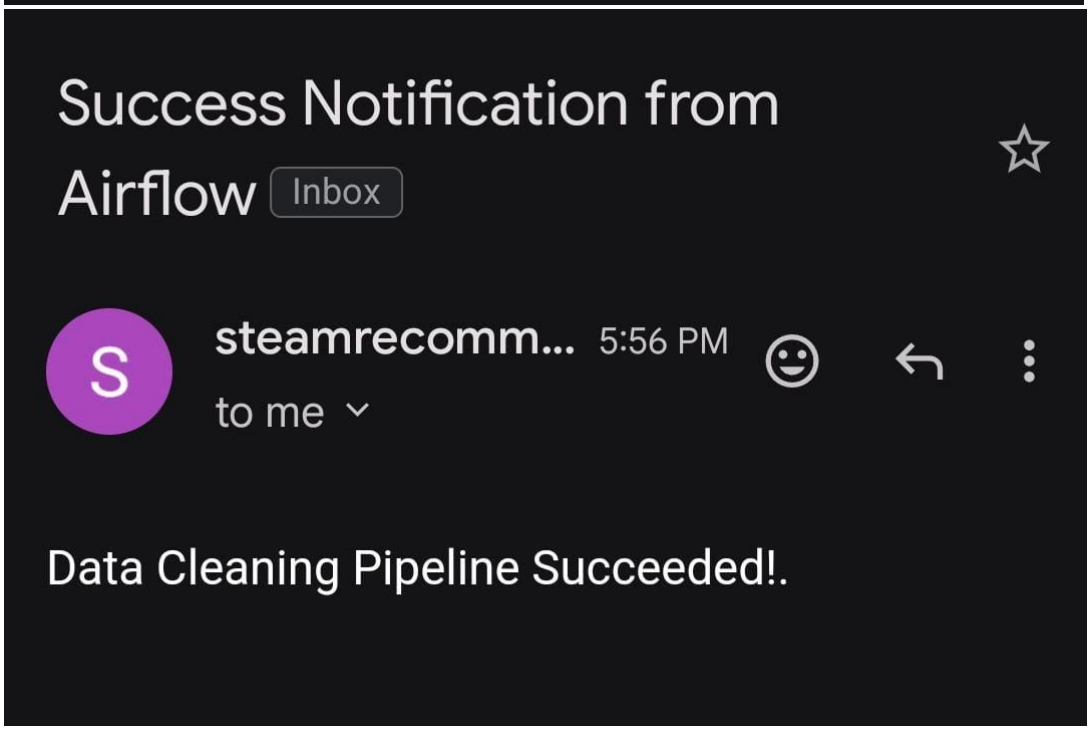
Implementation in DAG:

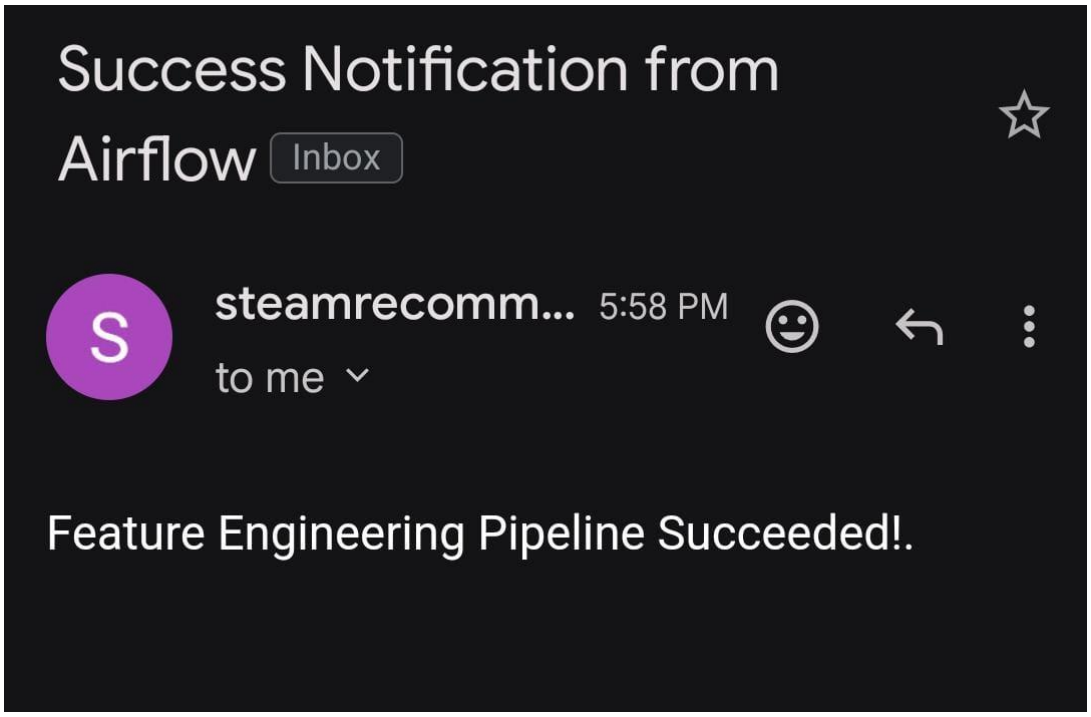
```
from notification import notify_success, notify_failure

task = PythonOperator(
    task_id='example_task',
    python_callable=my_function,
    on_success_callback=lambda context: notify_success(context, "Task completed
successfully."),
    on_failure_callback=lambda context: notify_failure(context, "Task execution
failed."),
    dag=dag,
```

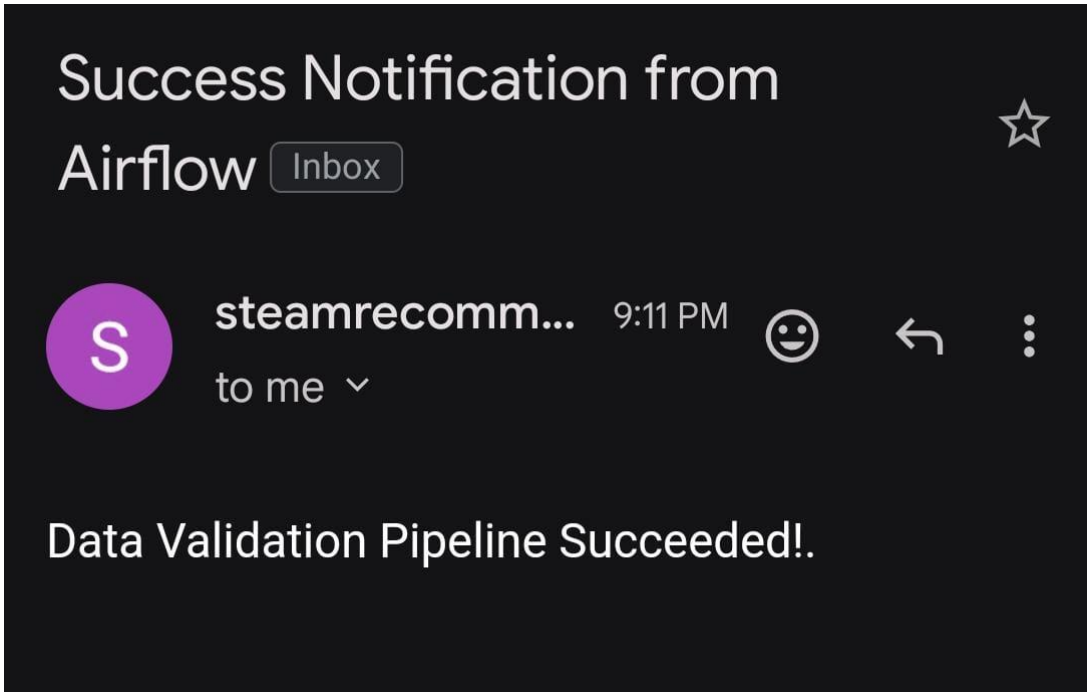
)

Images of notifications sent to our email after completion of each pipeline:






Success Notification from Airflow
Inbox
S
steamrecomm... 5:58 PM
to me
Feature Engineering Pipeline Succeeded!.


Success Notification from Airflow
Inbox
S
steamrecomm... 9:11 PM
to me
Data Validation Pipeline Succeeded!.