AI Chatbot Data-Pipeline Documentation

Group 1 - Team Members:

- Tirth Desai
- Priyankkumar Patel
- Jigar Naik

Project Overview

The **AI-Chatbot-Data-Pipeline** is designed to handle large-scale e-commerce data. Its purpose is to streamline the ETL (Extract, Transform, Load) processes for datasets such as Amazon user reviews and product metadata, transforming this data into a structured, query-optimized format in a data warehouse.

Contents of README Documentation

1. Instructions for Environment Setup

Prerequisites

- **Python**: Version 3.8+
- **Cloud Services**: Google Cloud Storage (GCS) for file storage
- **PostgreSQL**: For the data warehouse
- **Apache Airflow**: For orchestration

1. Setup Steps

These steps help prepare the environment, ensuring all dependencies are in place and configurations are set.

Cloning the Repository

```
git clone <repository-url>
cd eCom-Chatbot-Data-Pipeline
```

- **Explanation**:
    - **`git clone <repository-url>`**: This command copies the project repository from GitHub (or another version control platform) to your local machine.
    - **`cd eCom-Chatbot-Data-Pipeline`**: Changes the directory to the newly cloned repository, allowing you to work within the project folder.

Setting Up a Virtual Environment

```
python3 -m venv env
source env/bin/activate  # For Linux/macOS
env\Scripts\activate     # For Windows
```

- **Explanation**:
  - **`python3 -m venv env`**: Creates a virtual environment named `env`. A virtual environment is an isolated Python environment that keeps dependencies separate from global installations, preventing version conflicts.
  - **`source env/bin/activate`** (Linux/macOS) or **`env\Scripts\activate`** (Windows): Activates the virtual environment, making it the default Python environment for this session. This ensures that when you install packages, they're added only to this isolated environment, not system-wide.

Installing Dependencies

```
pip install -r requirements.txt
```

- **Explanation**:
  - **`pip install -r requirements.txt`**: Installs all Python packages listed in the `requirements.txt` file, which contains a list of dependencies required by the project. Each package's specific version is listed to avoid compatibility issues, ensuring consistency across different setups.

2. Setting Up Environment Variables

You'll create a `.env` file containing sensitive information like database credentials and API keys. An example `.env` file might look like this:

```
GCP_ACCESS_KEY_ID=<your-access-key>
GCP_SECRET_ACCESS_KEY=<your-secret-key>
DB_USER=<database-username>
DB_PASSWORD=<database-password>
GCS_BUCKET_NAME=<your-bucket-name>
```

- **Explanation**:
  - **Environment Variables**: This step configures secret credentials and access keys securely. Using environment variables keeps sensitive data out of the codebase, reducing security risks.
  - **Structure of `.env`**: Each variable is a key-value pair. For example, `AWS_ACCESS_KEY_ID` holds your Amazon Web Services access key. The code within the project will load these values to access cloud services and databases securely.

3. Steps to Run the Pipeline

The pipeline is orchestrated using Apache Airflow, which automates and schedules tasks. Below are the commands for initializing and launching Airflow.

1) **Initializing the Airflow Database**

```
airflow db init
```

- **Explanation for this line of code**:
  - **`airflow db init`**: Sets up the initial Airflow metadata database, which stores task statuses, configuration settings, and execution history. This is a one-time setup required before running any Airflow tasks.

2) **Trigger the ETL Pipeline DAG**:
   - In the Airflow web interface, we use ETL DAG to start data processing. The DAG organizes each task in sequence, handling data flow from raw extraction to loading in the database.

4. Starting the Scheduler and Web Server

```
airflow scheduler &
airflow webserver -p 8080
```

- **Explanation**:
  - **`airflow scheduler &`**: Starts the Airflow scheduler, which triggers tasks according to the schedules defined in the Airflow DAGs (Directed Acyclic Graphs). The `&` symbol runs the command in the background, allowing you to run other commands in the same terminal.
  - **`airflow webserver -p 8080`**: Launches the Airflow web interface on port 8080, making it accessible in a web browser at `http://localhost:8080`. This interface lets you monitor and control DAGs and view logs for each task.

5.Triggering the ETL Pipeline DAG in Airflow

Once the web server is running, open a browser and navigate to `http://localhost:8080`. Here, you'll find the DAG for the ETL pipeline.

- **Explanation**:
  - In the Airflow web interface, you can manually trigger the DAG by selecting it and clicking the "Trigger DAG" button. This starts the ETL process, where each task in the DAG executes sequentially or in parallel, based on dependencies.
  - **Monitoring and Logs**: The web interface displays the status of each task. You can view logs for each step to troubleshoot issues if a task fails, or check the progress as tasks complete successfully.

3. Code Structure and Explanation

This section provides an overview of each script, its purpose, and the logic behind the key functions.

3.1 Scripts Overview

- **`json_to_csv.py`**: Converts JSON files to CSV format and uploads them to Google Cloud Storage (GCS).
- **`download_data.py`**: Downloads and extracts data files from URLs, then uploads them to a GCS bucket.
- **`db_to_schema.py`**: Defines a database schema and migrates data from tables (such as `metadata` and `user_reviews`) into a structured format.
- **`db_connection.py`**: Connects to the PostgreSQL database using secure environment variables.
- **`bucket_connection.py`**: Establishes a connection to a GCS bucket for data upload and download operations.
- **`CSV_to_DB.py`**: Loads CSV data into database tables, ensuring the data structure aligns with the database schema.

3.2 Explanation of Code Logic

`json_to_csv.py`

- **Purpose**: Converts JSON files from GCS into CSV format for easier data handling in the database.
- **Key Function: `json_to_csv_meta(source_blob_name, destination_blob_directory)`**:
  - **Logic**: Downloads the JSON file, reads it into a DataFrame, converts it to CSV, and uploads it back to GCS.
  - **Why**: Converting JSON to CSV standardizes the data format, making it compatible with database loading processes.

`download_data.py`

- **Purpose**: Downloads data from an external source, extracts it if needed, and uploads it to GCS.
- **Key Function: `ingest_data_meta(file_url)`**:
  - **Logic**: Downloads a zip file, extracts it, and stores the files in a GCS bucket.
  - **Why**: Ensures that fresh data is always available for the ETL pipeline, stored in a secure, accessible location.

`db_to_schema.py`

- **Purpose**: Creates the database schema and migrates data into structured tables.
- **Key Function: `db_to_schema()`**:
  - **Logic**: Uses SQLAlchemy to create the target schema and transfers data from raw tables to organized schema tables.
  - **Why**: Organizing data in schemas optimizes it for analytical queries and supports consistent data structure.

`db_connection.py`

- **Purpose**: Establishes and manages a connection to the PostgreSQL database.
- **Key Function: `connect_with_db()`**:
  - **Logic**: Retrieves environment variables for credentials, establishes a connection, and returns a database engine.
  - **Why**: Centralizing the database connection logic makes it reusable for other scripts that interact with the database.

`bucket_connection.py`

- **Purpose**: Connects to a GCS bucket to handle file upload and download operations.
- **Key Function: `connect_to_bucket()`**:
  - **Logic**: Uses Google Cloud's `storage` library to authenticate and connect to the specified bucket.
  - **Why**: Simplifies file interactions with GCS, making it easy to move files to and from cloud storage.

`CSV_to_DB.py`

- **Purpose**: Loads CSV files into PostgreSQL tables, creating or updating tables as necessary.
- **Key Function: `create_table_user_review()`**:
  - **Logic**: Checks if the table exists, creates it if not, then loads data from CSVs into the table.
  - **Why**: Ensures data is readily accessible in a structured format, stored in a relational database for querying.

4. Reproducibility

Following the steps outlined in the README file enables anyone to seamlessly reproduce the entire code and pipeline setup, ensuring a consistent and accurate replication of the project environment and functionality.

5. Error Handling and Logging

Error Handling

Each script includes specific error-handling mechanisms to address potential issues.

- **API and Network Errors**: Scripts connecting to external APIs or cloud storage handle connection errors and provide retries or alerts as necessary.
- **Database Errors**: Scripts connecting to the database capture errors related to connectivity and handle them without interrupting the entire process.

Logging

Logging is implemented across scripts for tracking progress and troubleshooting.

- **Log Messages**: Informative logs capture major events like successful connections and data uploads.
- **Error Logs**: Errors are logged with details on where they occurred and the type of issue, aiding in debugging and pipeline maintenance.

EXPLANATION ON THE STEPS THAT ARE BEING MENTIONED IN README FILE:

1. Docker Setup

Docker allows you to create isolated environments for running applications, which is ideal for ensuring consistency across different setups. Here's how each part of the Docker setup works.

Step-by-Step Explanation:

1. **Dockerfile Creation**:
   - The `Dockerfile` is the blueprint for creating a Docker image. It includes the instructions for setting up the environment with all dependencies, libraries, and configuration required to run the pipeline.
   - It usually starts with a base image (e.g., Python) and includes commands to install packages, copy necessary files, and set up environment variables.
2. **Build the Docker Image**:
   - **Command**: `docker build -t pipeline_image .`
   - **Explanation**: This command compiles the Docker image based on the `Dockerfile`. The `-t pipeline_image` assigns a tag (name) to the image (`pipeline_image`), making it easier to reference later.
3. **Docker Compose**:
   - **Explanation**: Docker Compose is a tool for defining and managing multi-container Docker applications. Using a `docker-compose.yml` file, you define multiple services (e.g., Airflow scheduler, web server, database) that are needed for the pipeline.
   - The `docker-compose.yml` specifies how each service interacts with the others, their ports, volumes, and environment variables.
4. **Starting Services with Docker Compose**:
   - **Command**: `docker-compose up -d`
   - **Explanation**: This command launches all services defined in the `docker-compose.yml` file in detached mode (`-d`), which runs them in the background. It sets up and starts all necessary containers for the pipeline environment, including the Airflow scheduler, web server, and any databases.

2. Setting Up Email Notifications

Email notifications help keep track of the pipeline's status by sending alerts for task successes, failures, or retries.

Step-by-Step Explanation:

1. **Configuring SMTP Settings**:
   - o **Explanation**: In Airflow, you set up SMTP (Simple Mail Transfer Protocol) configurations within the `airflow.cfg` file or as environment variables.
   - o **Example Configuration**: The SMTP server, port, and login credentials are specified. For instance, using `smtp.gmail.com` for Gmail notifications and setting the SMTP port to `587`.
2. **Adding Notification Settings to Airflow DAGs**:
   - o **Explanation**: In the DAG (Directed Acyclic Graph) definitions, you specify `email_on_failure` and `email_on_retry` settings. These enable automatic emails if tasks fail or retry, keeping you informed of the pipeline's status.
   - o **Customizing Recipient**: The email recipient is defined in the DAG file, so notifications are sent to the appropriate user's email whenever there's an alert.

3. Running Docker and Initializing Airflow

Once Docker is set up, running the Airflow pipeline within Docker ensures all configurations and dependencies are isolated and consistent. Here's what each line in this process does.

Step-by-Step Explanation:

1. **Starting Docker Containers**:
   - o **Command**: `docker-compose up -d`
   - o **Explanation**: This command starts up the Docker containers defined in the `docker-compose.yml` file. Running in detached mode (`-d`) allows the terminal to remain free for other commands.
2. **Accessing Airflow Web Interface**:
   - o **URL**: `http://localhost:8080`
   - o **Explanation**: With the Docker container running, Airflow's web server is accessible at `localhost` on port `8080`. The web interface allows monitoring and managing DAGs, viewing task statuses, and accessing logs.
3. **Initializing the Airflow Database**:
   - o **Command**: `docker exec -it <container_name> airflow db init`
   - o **Explanation**: This initializes the metadata database within the running Airflow container. The `docker exec -it` command allows you to execute commands inside a running container. `airflow db init` sets up the database that tracks DAGs, task history, and other configurations.
4. **Creating Airflow Users**:

- **Command**: `docker exec -it <container_name> airflow users create ...`
- **Explanation**: Airflow requires user accounts for accessing the web interface. This command creates a new user with specified credentials and roles, enabling access to Airflow's dashboard and features.

4. Dataset Information

This section provides details about the data used in the pipeline, such as its origin, format, and characteristics.

1. **Source**:
    - **Explanation**: Defines where the dataset originates, e.g., a public dataset of Amazon reviews. Knowing the source helps in understanding the data structure and ensuring data credibility.
2. **Format**:
    - **Explanation**: Specifies the data format (e.g., JSON, CSV) and structure. This information is crucial for designing ETL steps and helps ensure correct parsing and transformation.
3. **Data Fields**:
    - **Explanation**: Lists key fields in the dataset (e.g., review text, rating), along with data types. Knowing the fields helps define transformations, enrichments, and schema design in the database.
4. **Volume**:
    - **Explanation**: Provides an estimate of data size (e.g., number of records, file size). This helps in optimizing storage, managing memory, and planning processing power.

5. Data Card

1. User Reviews Dataset

This dataset captures user-generated content, specifically reviews on various products. Each feature (column) represents a specific type of information about the reviews.

Explanation of Each Feature in the User Reviews Dataset

Feature Name	Feature Data Type	Feature Description
rating	Float	Represents the rating given by the user, usually ranging from 1 to 5. This helps in understanding the user's satisfaction level with the product.
title	String	The title of the user review. A short summary of the review text, providing a quick glimpse of the review's tone.

Feature Name	Feature Data Type	Feature Description
text	String	The full body of the user's review, capturing detailed feedback on the product. This field can be used for sentiment analysis or topic modeling.
images	List	A list of URLs or paths to images posted by the user. Images can provide additional context to the review.
asin	String	ASIN (Amazon Standard Identification Number), a unique identifier for the product. Used for linking reviews to specific products.
parent_asin	String	The parent product ID, useful for grouping related products (e.g., variations in size or color).
user_id	String	Unique ID of the reviewer, useful for analyzing individual user behaviors and reviewing patterns.
timestamp	Integer	The UNIX timestamp of the review submission. Useful for time-based analysis, such as tracking trends over time.
verified_purchase	Boolean	Indicates whether the user actually purchased the product. Helps in distinguishing genuine reviews from general feedback.
helpful_vote	Integer	The number of votes marking the review as helpful. Provides insight into the perceived value of the review by other users.

How the User Reviews Dataset is used

- **Sentiment Analysis**: Fields like `rating`, `title`, and `text` are essential for analyzing user sentiment and satisfaction levels with products.
- **Product Linking**: The `asin` and `parent_asin` fields link reviews to specific products, making it possible to associate user feedback with product metadata.
- **Trend Analysis**: The `timestamp` field allows for time-based trend analysis to track how user opinions and ratings evolve over time.
- **Authenticity**: The `verified_purchase` and `helpful_vote` fields help gauge the authenticity and usefulness of the reviews, potentially filtering out unhelpful or fake reviews.

2. Product Metadata Dataset

This dataset provides detailed information about the products listed on the e-commerce platform. Each feature describes a specific aspect of the product, such as category, rating, and price.

Explanation of Each Feature in the Product Metadata Dataset

Feature Name	Feature Data Type	Feature Description
main_category	String	High-level domain of the product, such as electronics, home goods, etc. Useful for categorizing and segmenting products.
title	String	The name of the product as displayed on the product page.
average_rating	Float	The product's average rating, shown on the product page, aggregated from all user ratings.
rating_number	Integer	Total number of ratings for the product. Useful for determining popularity and sales volume.
features	List	Key features of the product in bullet-point format. Helps in analyzing and comparing product attributes.
description	List	Detailed description of the product, often listing its specifications and benefits.
price	Float	Product price in US dollars, useful for pricing analysis and comparisons.
images	List	A collection of product images. Helpful for visual content analysis or enhancing the product listing with multimedia.
videos	List	Product-related videos, often including titles and URLs. Can be used for understanding marketing strategies or customer engagement.
store	String	The name of the store selling the product, important for vendor-specific analysis.
categories	List	Hierarchical categories of the product, providing context on the product's placement within the overall catalog.
details	Dictionary	Additional product details, such as materials, brand, and sizes. Used to enhance product descriptions and compare product specifications.
parent_asin	String	Parent ID for variations of the product, similar to the `User Reviews Dataset`. Useful for grouping product variations.
bought_together	List	Lists other products often bought together with this one, valuable for recommendation systems.

How the Product Metadata Dataset is used

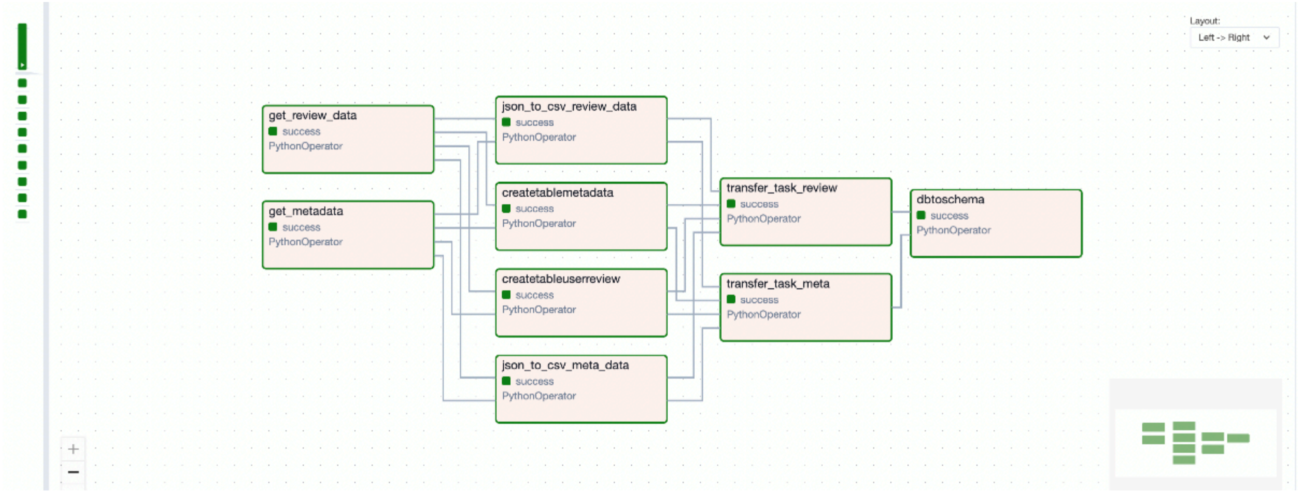
- **Product Analytics**: Fields like `price`, `rating_number`, and `average_rating` are useful for sales and pricing analysis, identifying popular or well-rated products.
- **Categorization and Segmentation**: The `main_category` and `categories` fields allow products to be organized into hierarchical categories, making it easy to filter and analyze products by segment.

- **Product Comparison**: The `features`, `description`, and `details` fields provide rich information for comparing products based on their specifications and attributes.
- **Recommendation System**: The `bought_together` field supports the creation of recommendation systems by identifying commonly bundled products.

5. Data Pipeline Components

The data card plays a crucial role in shaping the data pipeline, as each feature in the datasets guides the design of various processing tasks. Initially, the pipeline retrieves raw data from the source, ensuring that all necessary fields—such as `rating`, `main_category`, and `features`—are included for thorough analysis. This data comprises user reviews and product metadata, with each field selected to support future insights.Once the data is gathered, it is cleaned and organized, making it ready for structured storage. The processed data is then organized into a data warehouse with a schema that reflects the datasets' structure. For example, a dedicated table for user reviews is created to hold information such as `user_id`, `rating`, and `text`, while a separate table for product metadata includes details like `title`, `average_rating`, and `categories`. These tables are carefully structured to ensure efficient querying and facilitate deeper analysis. To enhance the connection between these datasets, a relationship is established by linking them through shared identifiers—specifically, the `asin` and `parent_asin` fields. This connection allows user reviews to be associated with their respective product details, enabling comprehensive insights into how customer feedback aligns with particular product attributes. By designing the pipeline in this way, the data becomes more valuable and accessible for further analysis, allowing for meaningful conclusions about customer experiences and product performance.

6. **DATA SOURCE**



Explanation of Each Task

1. **get_review_data** (PythonOperator):
   - **Purpose**: This task likely retrieves the **User Reviews** data from an external source (e.g., an API or cloud storage).
   - **Outcome**: Once the data is successfully fetched, it's ready for further processing.
2. **get_metadata** (PythonOperator):
   - **Purpose**: This task retrieves the **Product Metadata** from an external source, similar to `get_review_data`.
   - **Outcome**: After fetching, the product metadata is available for transformation and loading.
3. **json_to_csv_review_data** (PythonOperator):
   - **Purpose**: Converts the **User Reviews** data from JSON format to CSV. Converting to CSV can make it easier to load into a relational database.
   - **Dependency**: This task depends on `get_review_data` and only starts once the review data has been successfully fetched.
4. **json_to_csv_meta_data** (PythonOperator):
   - **Purpose**: Similar to `json_to_csv_review_data`, this task converts the **Product Metadata** from JSON format to CSV.
   - **Dependency**: This task depends on `get_metadata` and begins only after the metadata has been fetched.
5. **createtablemetadata** (PythonOperator):
   - **Purpose**: Creates the database table for storing **Product Metadata** in the target database if it doesn't exist.
   - **Dependency**: This task depends on both `json_to_csv_review_data` and `json_to_csv_meta_data`, indicating that it requires the data in CSV format before setting up the table.
6. **createtableuserreview** (PythonOperator):
   - **Purpose**: Creates the table for storing **User Reviews** data in the target database.
   - **Dependency**: Similar to `createtablemetadata`, this task depends on the completion of the CSV conversion tasks for both datasets.
7. **transfer_task_review** (PythonOperator):
   - **Purpose**: Transfers the **User Reviews** CSV data into the newly created database table.
   - **Dependency**: This task depends on the `createtableuserreview` task, meaning the table must exist before data transfer occurs.
8. **transfer_task_meta** (PythonOperator):
   - **Purpose**: Transfers the **Product Metadata** CSV data into the database table created by `createtablemetadata`.
   - **Dependency**: This task also requires the metadata table to be created before data transfer begins.

9. **dbtoschema** (PythonOperator):

  - **Purpose**: Finalizes the database by setting up schemas for both datasets, organizing the data for optimized querying and storage.
  - **Dependency**: This task depends on both `transfer_task_review` and `transfer_task_meta`, indicating that it starts only after the data for both reviews and metadata has been successfully transferred to the database.

Workflow Summary

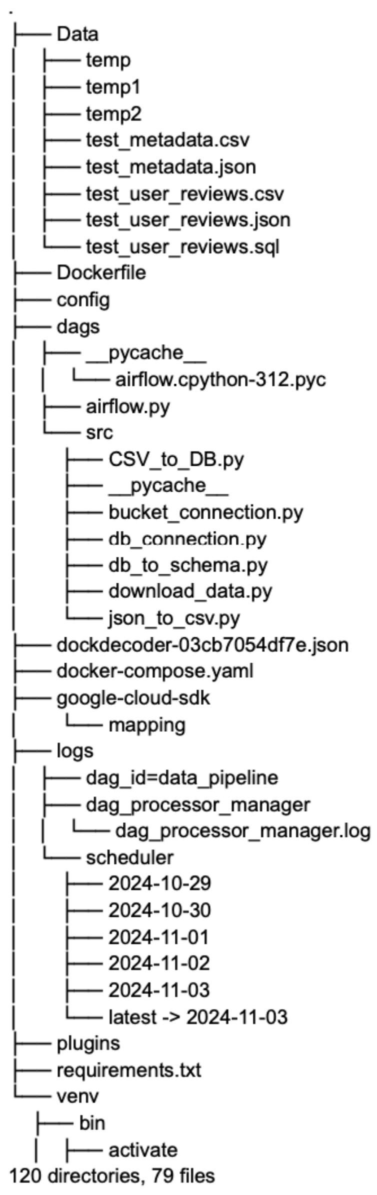
This pipeline performs the following sequence of actions:

1. **Data Retrieval**:
   - The pipeline starts by fetching the **User Reviews** (`get_review_data`) and **Product Metadata** (`get_metadata`) from external sources.
2. **Data Transformation**:
   - Once the data is fetched, it's converted from JSON to CSV by `json_to_csv_review_data` and `json_to_csv_meta_data`. This step standardizes the data format, preparing it for loading into a relational database.
3. **Table Creation**:
   - After the data is in CSV format, the pipeline creates tables for both datasets (`createtableuserreview` and `createtablemetadata`). These tables are structured to store the CSV data in an organized way.
4. **Data Transfer**:
   - The CSV data is then transferred into the respective database tables using `transfer_task_review` and `transfer_task_meta`.
5. **Schema Finalization**:
   - Finally, `dbtoschema` sets up the necessary database schema, organizing both tables for efficient querying and storage.

Key Points

- **Parallel Processing**: `get_review_data` and `get_metadata` can run in parallel since they do not depend on each other.
- **Sequential Dependencies**: Each step has dependencies, meaning certain tasks must complete successfully before others begin. For instance, the table creation tasks (`createtableuserreview` and `createtablemetadata`) depend on the data conversion to CSV.
- **Final Schema Setup**: The `dbtoschema` task is the last step and ensures the data is organized and accessible, finalizing the pipeline.

Folder Structure



```
.
├── Data
│   ├── temp
│   ├── temp1
│   ├── temp2
│   ├── test_metadata.csv
│   ├── test_metadata.json
│   ├── test_user_reviews.csv
│   ├── test_user_reviews.json
│   └── test_user_reviews.sql
├── Dockerfile
├── config
├── dags
│   ├── __pycache__
│   │   └── airflow.cpython-312.pyc
│   ├── airflow.py
│   └── src
│       ├── CSV_to_DB.py
│       ├── __pycache__
│       ├── bucket_connection.py
│       ├── db_connection.py
│       ├── db_to_schema.py
│       ├── download_data.py
│       └── json_to_csv.py
├── dockdecoder-03cb7054df7e.json
├── docker-compose.yaml
├── google-cloud-sdk
│       └── mapping
├── logs
│   ├── dag_id=data_pipeline
│   ├── dag_processor_manager
│   │   └── dag_processor_manager.log
│   └── scheduler
│       ├── 2024-10-29
│       ├── 2024-10-30
│       ├── 2024-11-01
│       ├── 2024-11-02
│       ├── 2024-11-03
│       └── latest -> 2024-11-03
├── plugins
├── requirements.txt
└── venv
    ├── bin
    │   ├── activate
120 directories, 79 files
```

1. Data Directory

The `Data` directory contains raw data files and temporary directories used in data processing.

- **temp, temp1, temp2**: These are temporary folders likely used for staging data during various transformations. For example, they might store intermediate files (e.g., unzipped data) before the data is processed further or uploaded.

- **test_metadata.csv / test_metadata.json**: Sample files containing product metadata in both CSV and JSON formats, which are probably used for testing data ingestion and transformation functions.
- **test_user_reviews.csv / test_user_reviews.json**: Sample files containing user reviews data in CSV and JSON formats, allowing you to test the pipeline with mock data.
- **test_user_reviews.sql**: This could be a SQL script or a mock database export of user reviews, useful for testing database-related operations without relying on external data sources.

2. Dockerfile

The `Dockerfile` defines the instructions for building a Docker image of this project. It includes details such as the base image, dependencies, and environment configurations. Using Docker ensures that the application runs in a consistent environment, regardless of the machine it's running on.

3. config Directory

This directory typically stores configuration files, such as environment settings or API keys. Although the specific files are not shown here, the `config` directory might contain `.env` files or other sensitive configurations that are necessary for connecting to databases, cloud storage, or other services.

4. dags Directory

The `dags` directory contains the Airflow DAG (Directed Acyclic Graph) files, which define the workflow for the ETL pipeline. The structure here includes:

- **airflow.py**: This is likely the main DAG file that defines the workflow for the ETL pipeline in Airflow. It specifies the sequence of tasks and their dependencies.
- **src** Directory within `dags`: This directory contains the Python scripts that are used as tasks within the DAG.
  - **CSV_to_DB.py**: Handles loading data from CSV files into the database.
  - **bucket_connection.py**: Manages connections to cloud storage, such as Google Cloud Storage.
  - **db_connection.py**: Contains functions to connect to the database.
  - **db_to_schema.py**: Responsible for creating the database schema and possibly transforming or organizing data within the database.
  - **download_data.py**: Script for downloading data from external sources or APIs.
  - **json_to_csv.py**: Converts JSON files into CSV format for easier database ingestion.

These scripts work together to perform the steps in the ETL process and are called within the Airflow DAG (`airflow.py`) to execute in sequence.

5. dockdecoder-03cb7054df7e.json

This file appears to be a JSON key file for a service account, most likely for Google Cloud Platform (GCP). It contains the credentials needed to authenticate with GCP services, such as Google Cloud Storage or BigQuery. This file is essential for secure access to cloud resources and should be protected.

6. docker-compose.yaml

This file defines the configuration for Docker Compose, allowing you to run multiple services in a single command. It likely includes configurations for setting up the Airflow scheduler, web server, and other services needed for the pipeline. Docker Compose simplifies the setup by managing multiple containers with one command (`docker-compose up`).

7. google-cloud-sdk Directory

The `google-cloud-sdk` directory contains tools for interacting with Google Cloud services. The `mapping` folder within this directory might contain configuration files or scripts to map GCP resources, making it easier to manage services or automate tasks within Google Cloud.

8. logs Directory

This directory stores log files for Airflow, which are crucial for monitoring and troubleshooting the pipeline.

- **dag_id=data_pipeline**: A folder dedicated to logs for a specific DAG, named `data_pipeline`. Each task's execution logs would be saved here.
- **dag_processor_manager**: Contains logs for the DAG processor, which manages the parsing and loading of DAGs in Airflow. `dag_processor_manager.log` records the activity of the processor.
- **scheduler**: Stores logs for the Airflow scheduler, which orchestrates the execution of tasks based on their schedules and dependencies. Each date (e.g., `2024-10-29`, `2024-11-01`) represents a log folder for that day, allowing for daily tracking of scheduler activities.

9. plugins Directory

This directory is used to store custom plugins for Airflow. Plugins can extend Airflow's functionality with custom operators, hooks, sensors, or interfaces. Although specific plugins are not listed here, this folder would hold any custom Airflow components needed by the pipeline.

10. requirements.txt

This file lists the Python dependencies required by the project, including packages for Airflow, database connectors, cloud SDKs, and data processing libraries. Running `pip install -r requirements.txt` installs all required libraries, ensuring the environment has everything needed to run the pipeline.

11. venv Directory

The `venv` directory contains the virtual environment for the project. It includes installed packages and dependencies, keeping them isolated from the global Python environment. This directory allows for a consistent and reproducible environment for the project.

- **bin/activate**: A script to activate the virtual environment, making the project's dependencies accessible in the current shell session.

Summary

This folder structure is well-organized and modular, making it easy to manage, understand, and extend. Here's a summary of each component's purpose:

- **Data**: Contains sample data and temporary files for testing and processing.
- **Dockerfile & docker-compose.yaml**: Define the Docker setup for consistent, containerized execution of the pipeline.
- **config**: Stores configuration files, possibly including environment variables and credentials.
- **dags**: Holds the DAG definitions and task scripts for Airflow, specifying the ETL workflow.
- **logs**: Maintains logs for monitoring and debugging Airflow's activities.
- **plugins**: Allows for custom extensions to Airflow's functionality.
- **requirements.txt**: Lists required Python libraries for the project.
- **venv**: Provides an isolated virtual environment for dependencies.

6. Steps included in generating a GCP JSON Connection File

1. Create a Service Account in Google Cloud

- **Explanation**:
  - A **service account** is a special kind of Google account that belongs to your application or a virtual machine, not an individual user. It allows secure, automated access to GCP services.
  - **Purpose**: Creating a service account specifically for your application keeps access scoped only to the resources it needs, improving security by isolating permissions.

2. Assign Roles and Permissions

- **Explanation**:
  - When creating the service account, assign the roles required by your application. For example, if you need access to Google Cloud Storage, you might assign the **Storage Object Admin** role.
  - **Purpose**: Granting only the necessary permissions follows the principle of least privilege, reducing security risks by limiting what the service account can access.

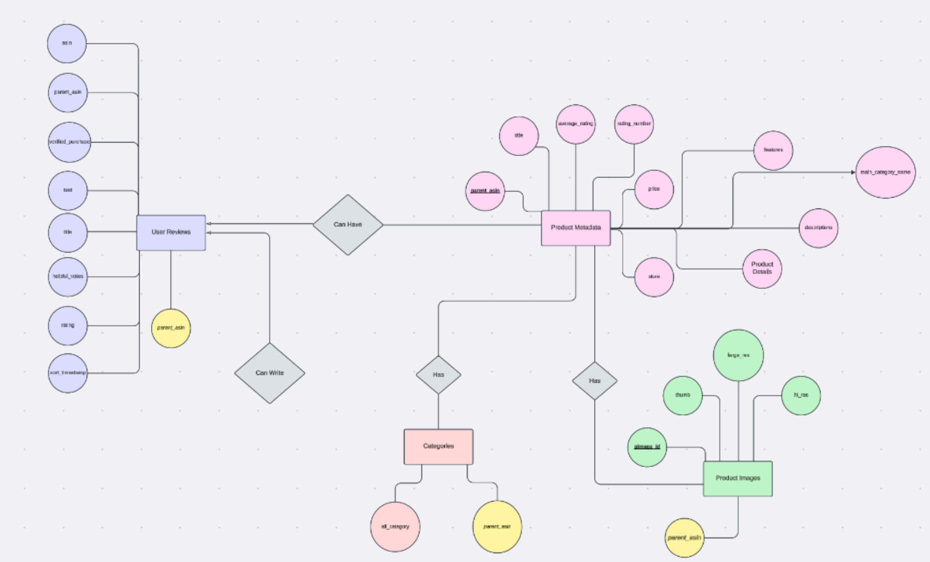
3. Generate and Download the JSON Key File

- **Explanation**:
  - In the GCP Console, after creating the service account, you can generate a **key** in JSON format. This file includes information such as the private key, client email, and project ID, all of which are required for authentication.
  - Click **Manage Keys** > **Add Key** > **Create New Key**, selecting **JSON** format. This will download a `.json` file to your local system.
  - **Purpose**: This JSON file serves as the credentials file that your application will use to authenticate with GCP services. It contains all the details needed to identify and authenticate the service account.

7. Usage of the JSON Key File in our Application

1. **Store the File Securely**:
   - **Explanation**: The JSON key file is sensitive information, as it allows access to GCP services. It should be stored securely (not in the codebase) and with restricted permissions to prevent unauthorized access.
   - **Purpose**: Ensuring that only trusted applications and users can access the key helps prevent potential misuse of your GCP account.
2. **Set the `GOOGLE_APPLICATION_CREDENTIALS` Environment Variable**:
   - **Command**: `export GOOGLE_APPLICATION_CREDENTIALS="/path/to/keyfile.json"`
   - **Explanation**: This environment variable tells the Google Cloud SDK and libraries where to find the JSON credentials file.
   - **Purpose**: Setting this variable allows applications and libraries to automatically detect and use the JSON key for GCP authentication.
3. **Use the JSON Key in Code**:
   - **Explanation**: Google Cloud libraries (like `google-cloud-storage` or `google-cloud-bigquery`) will automatically authenticate using the JSON credentials file if the environment variable is set.
   - **Purpose**: This setup enables seamless, secure access to GCP services without hardcoding credentials in the code.

8. SCHEMA



Key Entities and Relationships

1. **User Reviews** (Purple Box on the Left)

- **Attributes**:
  - `asin`: Amazon Standard Identification Number, a unique identifier for each product.
  - `parent_asin`: The ID of the parent product, useful for grouping similar products.
  - `verified_purchase`: Indicates whether the review was made by a verified purchaser, ensuring the review's authenticity.
  - `text`: The body of the user's review, providing feedback on the product.
  - `title`: A brief title or summary of the review.
  - `helpful_votes`: The number of users who found this review helpful.
  - `rating`: The user's rating of the product, often a number from 1 to 5.
  - `sort_timestamp`: The time when the review was posted, useful for time-based analysis.
- **Relationships**:
  - **Can Write**: This relationship indicates that users can write multiple reviews on various products.
  - **Can Have**: This relationship links `User Reviews` with `Product Metadata`, showing that reviews are associated with specific products using the `asin` and `parent_asin` attributes.

2. **Product Metadata** (Pink Box in the Center)
   - **Attributes**:
     - `title`: The name of the product.
     - `average_rating`: The average rating of the product based on all user reviews.
     - `rating_number`: The total number of ratings the product has received, indicating its popularity.
     - `price`: The price of the product, often in USD.
     - `features`: Key features of the product, typically in bullet-point format.
     - `main_category_name`: The primary category to which the product belongs, helping in classification.
     - `descriptions`: Detailed descriptions of the product's specifications and features.
     - `store`: The name of the store or vendor selling the product.
     - `parent_asin`: Used to connect products that are variations of the same item (e.g., different colors or sizes).

- **Relationships**:
    - **Can Have**: This relationship connects `Product Metadata` with `User Reviews`, showing that a product can have multiple reviews.
    - **Has (Categories)**: Links `Product Metadata` to the `Categories` entity, showing that each product belongs to specific categories.
    - **Has (Product Details)**: Represents that the product has additional details, such as a detailed description or features.
    - **Has (Product Images)**: Indicates that each product can have multiple images associated with it, helping users visually evaluate the product.

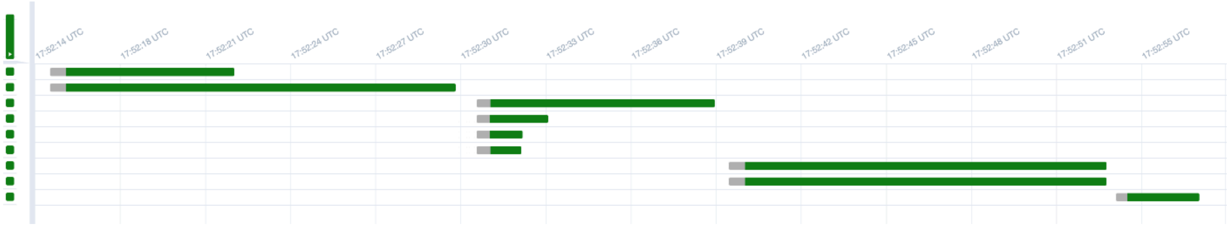
3. **Categories** (Red Box at the Bottom)
    - **Attributes**:
        - `all_category`: Represents the hierarchical categories to which the product belongs (e.g., Electronics > Mobile > Accessories).
        - `parent_asin`: Connects categories back to their parent products, allowing for hierarchical organization of products by category.
    - **Relationships**:
        - **Has**: Links `Product Metadata` with `Categories`, showing that each product can belong to multiple categories.
4. **Product Images** (Green Box on the Right)
    - **Attributes**:
        - `image_id`: A unique identifier for each image.
        - `thumb`: Thumbnail-sized image of the product, suitable for quick previews.
        - `large_res`: High-resolution version of the image, providing detailed visuals.
        - `hi_res`: Highest resolution image, often used for detailed product views.
    - **Relationships**:
        - **Has**: Connects `Product Metadata` to `Product Images`, indicating that each product can have multiple associated images, which might be in different resolutions.

Summary of Relationships

1. **User Reviews ↔ Product Metadata**:
    - Each product can have multiple reviews (indicated by the `Can Have` relationship). This relationship uses `asin` as the linking attribute, enabling efficient retrieval of all reviews related to a product.

2. **Product Metadata ↔ Categories**:
    - Products are classified into one or more categories (`Has` relationship). This allows for hierarchical organization, making it easy to filter products by category in queries.
3. **Product Metadata ↔ Product Images**:
    - Each product can have multiple images (`Has` relationship), stored with different resolutions (e.g., thumbnail, large, high-resolution). This relationship enhances the product listings by providing visual content.

GHANTT CHART:



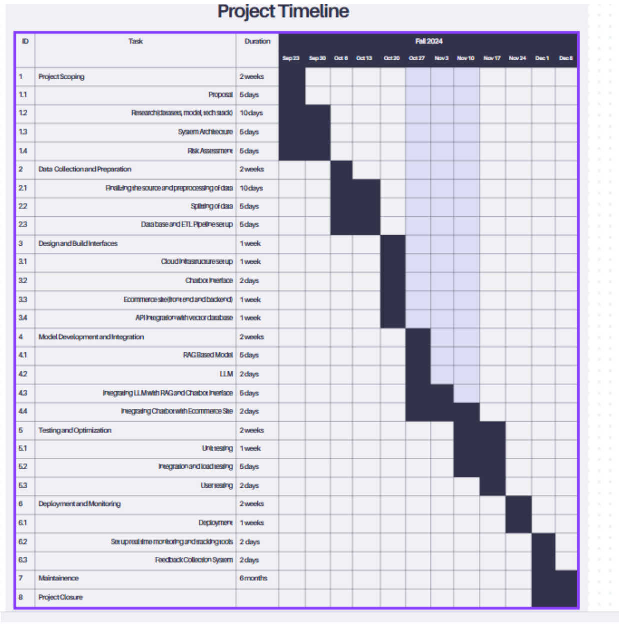
Explanation of the Chart

- **X-Axis (Time)**: The timeline, marked in UTC, shows when each task started and ended.
- **Y-Axis (Tasks)**: Each row represents a different task in the DAG (Directed Acyclic Graph), with the tasks arranged in the sequence of execution.
- **Green Bars**: Each green bar represents the execution duration of a task. Longer bars indicate tasks that took more time to complete.
- **Gray Bars**: The gray sections at the start of some tasks show the time the task was in a queued or waiting state before execution began.

Observations

1. **Parallel Execution**: Some tasks start and run in parallel, as seen with multiple bars aligned vertically, indicating that the DAG is optimized to execute tasks concurrently where possible.
2. **Sequential Dependencies**: Some tasks only start after their dependencies complete. For example, tasks on the left side start first and are followed by the next set, showing dependencies within the pipeline.
3. **Completion Times**: Most tasks complete within a close time frame, showing efficient management of the pipeline.

GHANTT CHART 2:



Project Timeline

ID	Task	Duration	Fall 2024											
			Sep 23	Sep 30	Oct 6	Oct 13	Oct 20	Oct 27	Nov 3	Nov 10	Nov 17	Nov 24	Dec 1	Dec 8
1	Project Scoping	2 weeks	■											
1.1	Proposal	5 days	■											
1.2	Research(datasets, model, tech stack)	10 days	■	■										
1.3	System Architecture	5 days	■	■										
1.4	Risk Assessment	5 days	■	■										
2	Data Collection and Preparation	2 weeks			■									
2.1	Finalizing the source and preprocessing of data	10 days			■	■								
2.2	Splitting of data	5 days			■	■								
2.3	Data base and ETL Pipeline set up	5 days			■	■								
3	Design and Build Interfaces	1 week					■							
3.1	Cloud Infrastructure set up	1 week					■							
3.2	Chatbot Interface	2 days					■							
3.3	Ecommerce site(front end and backend)	1 week					■							
3.4	API Integration with vector database	1 week					■							
4	Model Development and Integration	2 weeks						■						
4.1	RAG Based Model	5 days						■						
4.2	LLM	2 days						■						
4.3	Integrating LLM with RAG and Chatbot Interface	5 days						■	■					
4.4	Integrating Chatbot with Ecommerce Site	2 days						■	■	■				
5	Testing and Optimization	2 weeks								■	■			
5.1	Unit testing	1 week								■	■			
5.2	Integration and load testing	5 days								■	■			
5.3	User testing	2 days									■			
6	Deployment and Monitoring	2 weeks										■		
6.1	Deployment	1 weeks										■		
6.2	Set up real time monitoring and tracking tools	2 days											■	
6.3	Feedback Collection System	2 days											■	
7	Maintainence	6 months											■	■
8	Project Closure												■	■

This image shows a **Project Timeline** in the form of a Gantt chart, detailing the sequential and overlapping phases of a project, likely related to a data pipeline or system integration project.

Key Elements

- **Tasks**: Each row represents a specific task or milestone within the project, organized from start to finish. Tasks include activities such as "Project Scoping," "Data Collection and Preparation," "Model Development and Integration," "Testing and Optimization," and "Deployment and Monitoring."
- **Duration**: The second column lists the duration for each task (e.g., days, weeks), showing how long each step is expected to take.
- **Timeline**: The horizontal bars represent the timeline, marked by weeks or months across the top, indicating when each task starts and ends. Darker bars signify active periods for each task.
- **Phases**: The chart shows overlapping tasks, indicating concurrent work on different aspects of the project. For example, "Data Collection" might run alongside "Design and Build" phases.

Summary

This Gantt chart provides a high-level overview of the project schedule, showing the dependencies and timing of tasks. It helps track project progress and ensures that all phases—from planning and data preparation to deployment and maintenance—are completed within the planned time frame.