# Promptly Data Pipeline Report

## 1. Overview

The **Promptly Data Pipeline** is a structured workflow designed to process **user queries and documents (PDF, TXT files)** for an **AI-driven document-based Q&A system**. The system uses **Retrieval-Augmented Generation (RAG)** to fetch relevant answers based on user queries.

This pipeline is built using **Apache Airflow for orchestration, Supabase for database management, Google Cloud Storage (GCS) for document storage, and Data Version Control (DVC) for dataset tracking**. The pipeline follows **MLOps best practices** to ensure **data consistency, versioning, and automation**.

This report details all pipeline components, architecture, and justifications for any sections that are not applicable.

---

## 2. Key Components in the Data Pipeline

The data pipeline follows a structured approach, covering **data acquisition, preprocessing, versioning, orchestration, tracking, and logging**.

### 2.1 Data Acquisition

**Data Sources:**

- **User Queries**: Retrieved from the `conversations` table in **Supabase**.
- **Documents**: Uploaded by users and stored in **Google Cloud Storage (GCS)**.

**Implementation:**

- Queries are fetched using the `supabase_utils.py` script.
- Documents are retrieved from the `promptly-chat` bucket in GCS.
- All dependencies and external configurations are managed in `requirements.txt`.

**Reproducibility:**

- Data retrieval steps are automated through Airflow DAGs.
- Dependencies and configurations are versioned using **DVC** to ensure replicability.

## 2.2 Data Preprocessing

**Query Processing:**

- **Schema validation** using `validate_schema.py in both rag and user queries pipeline.`.
- **Text cleaning** using `data_utils.py` (stopword removal, special character removal, lowercasing).
- **Tokenization and Lemmatization** using `NLTK` and `WordNetLemmatizer`.

**Document Processing:**

- **Personally Identifiable Information (PII) Detection and Redaction** using **Presidio-based Named Entity Recognition (NER)** in `check_pii_data.py`.
- **Text Chunking** using `MarkdownSplitter` in `rag_utils.py`.

**Output:**

- Preprocessed **user queries and documents** are stored in **Supabase and GCS**.

## 2.3 Test Modules

**Unit Testing Frameworks:**

- `pytest`
- `unittest`

**Implemented Test Cases:**

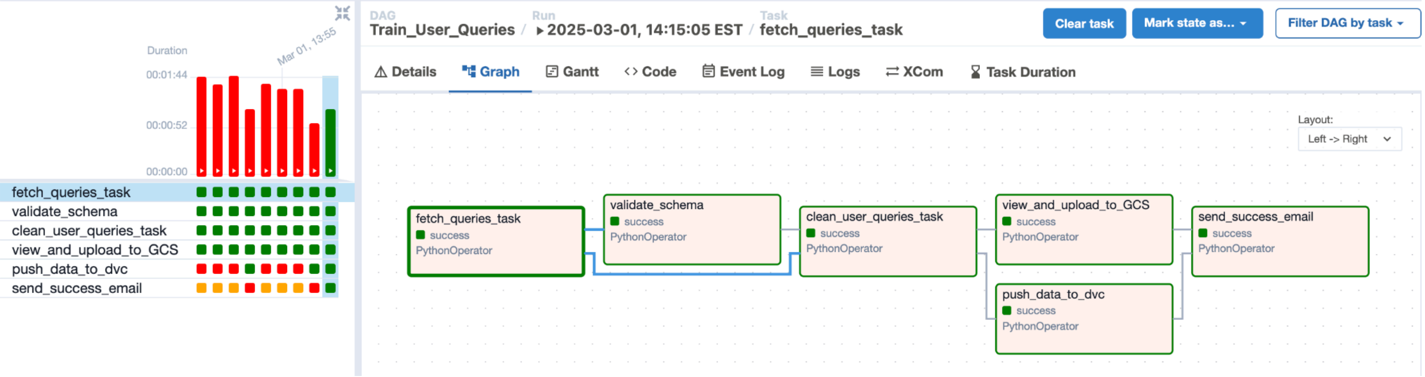
1. **PII Detection and Redaction** (`test_data_pii_redact_test.py`)
2. **RAG Pipeline** (`test_rag_pipeline.py`)
3. **User Query Pipeline** (`test_user_queries.py`)

## 2.4 Pipeline Orchestration (Airflow DAGs)

### User Queries DAG: Train_User_Queries

**Workflow:**

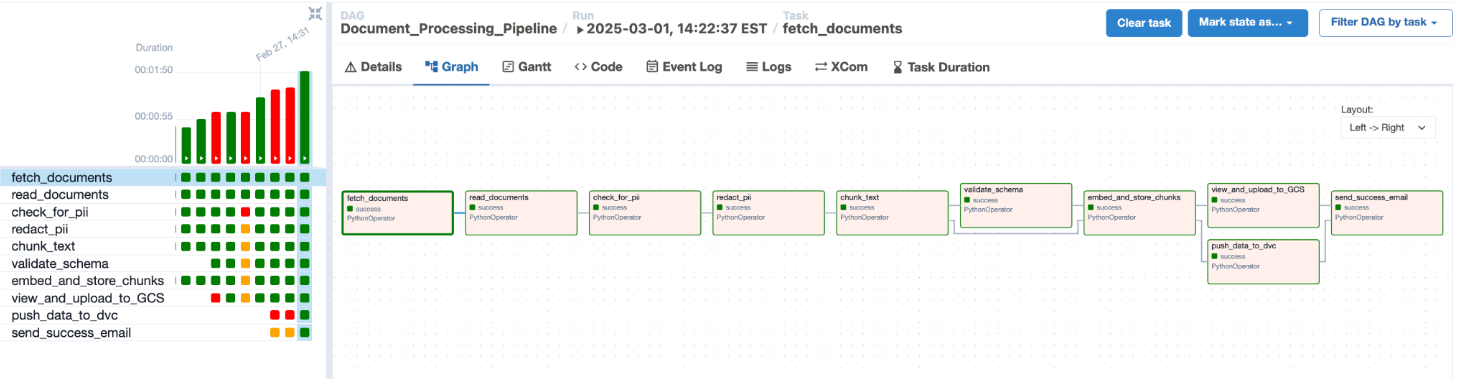
1. `fetch_queries_task`: Retrieves queries from Supabase.
2. `validate_schema`: Ensures schema consistency.
3. `clean_user_queries_task`: Cleans and preprocesses text.
4. `view_and_upload_to_GCS`: Saves processed queries to **GCS**.
5. `push_data_to_dvc`: Enables versioning via **DVC**.
6. `send_success_email`: Sends completion notifications.



### Document Processing DAG: Document_Processing_Pipeline

**Workflow:**

1. `fetch_documents`: Retrieves uploaded PDFs/TXT files.
2. `read_documents`: Extracts text using `pymupdf4llm`.
3. `check_for_pii`: Identifies PII using **Presidio**.
4. `redact_pii`: Redacts sensitive data.
5. `chunk_text`: Splits documents into structured chunks.
6. `validate_schema`: Ensures correct formatting.
7. `embed_and_store_chunks`: Generates embeddings using `Nomic` and stores them in **Supabase**.
8. `view_and_upload_to_GCS`: Stores chunked documents in **GCS**.
9. `push_data_to_dvc`: Enables versioning via **DVC**.
10. `send_success_email`: Sends completion notifications.



DAG Document_Processing_Pipeline / Run 2025-03-01, 14:22:37 EST / Task fetch_documents

Duration
00:01:50
00:00:55
00:00:00
Feb 27, 14:31
fetch_documents
read_documents
check_for_pii
redact_pii
chunk_text
validate_schema
embed_and_store_chunks
view_and_upload_to_GCS
push_data_to_dvc
send_success_email
Clear task
Mark state as...
Filter DAG by task
Details
Graph
Gantt
Code
Event Log
Logs
XCom
Task Duration
Layout:
Left -> Right
fetch_documents success PythonOperator
read_documents success PythonOperator
check_for_pii success PythonOperator
redact_pii success PythonOperator
chunk_text success PythonOperator
validate_schema success PythonOperator
embed_and_store_chunks success PythonOperator
view_and_upload_to_GCS success PythonOperator
push_data_to_dvc success PythonOperator
send_success_email success PythonOperator

---

## 2.5 Data Versioning with DVC

**Implementation:**

- **User Queries**: Versioned using **DVC** (`preprocessed_user_queries.csv.dvc`).
- **Processed Documents**: Versioned using **DVC** (`preprocessed_docs_chunks.csv`).
- **GCS Remote Storage**: Configured in `.dvc/config`.

**DVC Workflow:**

```
dvc init

dvc remote add gcs_remote gs://promptly-chat

dvc push
```

**Benefits of Using DVC:**

- Ensures **reproducibility** for dataset modifications.
- Allows **rollback** to previous versions.
- Provides **data lineage tracking** for query processing and document embedding versions.

---

## 2.6 Tracking and Logging

**Logging Frameworks Used:**

- **Airflow DAG Logs**
- **Python Logging Module**
- **Supabase Database Logging**

**Implementation Example:**

```
import logging

logging.basicConfig(level=logging.INFO)
```

**Error Handling Strategies:**

- **Retries in Airflow DAGs**
- **Failure Notifications via Email**
- **Exception Handling in Scripts**

---

## 2.7 Data Schema and Statistics Generation

**Implementation:**

- **Schema Validation**: Implemented in `validate_schema.py` before storing data in Supabase.
- **Statistics Generation**: Supabase logs track query/document statistics.

**Justification for Not Using TFDV or MLMD:**

- The system is **not an ML training pipeline**; it is focused on **retrieval**.
- Instead of **TensorFlow Data Validation (TFDV)**, schema checks are **embedded in preprocessing scripts**.

---

## 2.8 Anomaly Detection and Alerts

**Implemented Anomaly Detection:**

- **Missing Data Checks**: Handled in `validate_schema.py`.
- **Unexpected Formats Detection**: Managed in schema as well as in `data_utils.py`.

**Anomaly Alerts / Notifications:**

- The system operates **on batch processing**, not **real-time streaming data**.
- Pipeline failures are monitored through **Airflow Logs and email notifications**.

---

## 3. Data Bias Detection Using Data Slicing

**Justification for Not Implementing Bias Detection:**

- Our chatbot processes IT-related documents stored in a Retrieval-Augmented Generation (RAG) system, which contains purely technical content free from demographic or subjective influences.
- No Demographic Features: Our data lacks attributes like age, gender, or location, making subgroup analysis irrelevant.
- No Subjective Bias: The dataset consists of factual IT documentation, ensuring objectivity in responses.
- Unnecessary Overhead: Bias detection tools like TFMA or Fairlearn are designed for datasets with social implications. Applying them here would add complexity without meaningful insights.
- Since bias is not an inherent risk in our dataset, we will be focusing on enhancing precision and relevance in chatbot responses.

---

## 4. Folder Structure and Code Organization

The project follows **MLOps best practices** for modularity and scalability.

```
├── assets/
│   ├── process_user_queries_dag.png  # User Query Pipeline DAG
│   ├── rag_data_pipeline_dag.png  # Data Pipeline Workflow Diagram
│
├── data_pipeline/
│   ├── dags/
│   │   ├── dataPipeline.py  # User Queries DAG
│   │   ├── rag_data_pipeline.py  # Document Processing DAG
│   │   ├── scripts/
│   │   │   ├── email_utils.py  # Email notifications
│   │   │   ├── upload_data_GCS.py  # GCS Uploading
│   │   │   ├── data_preprocessing/
│   │   │   │   ├── check_pii_data.py  # PII Detection
│   │   │   │   ├── validate_schema.py  # Schema Validation
│   │   │   │   ├── data_utils.py  # Query Cleaning Functions
│   │   │   ├── supadb/
│   │   │   │   ├── supabase_utils.py  # Supabase Integration
│   │   │   ├── rag/
│   │   │   │   ├── rag_utils.py  # Chunking & Embeddings
│   │   ├── tests/
│   │   │   ├── test_data_pii_redact.py  # Tests for PII detection
│   │   │   ├── test_rag_pipeline.py  # Tests for RAG doc pipeline
│   │   │   ├── test_user_queries.py # Tests for User Queries pipeline
│   ├── config.py  # API Keys & Configurations
│   ├── README.md  # Project Documentation
│
├── data/
│   ├── rag_documents/  # Original PDFs & Text Files
│   ├── preprocessed_docs_chunks.csv  # Cleaned & Chunked Data
│   ├── preprocessed_user_data.csv  # Processed Queries
│
```

```
├── .dvc/  # DVC Configuration

├── .gitignore

├── requirements.txt  # Dependencies
```

---

## 5. Replicability and Reproducibility

**Steps to Set Up:**

1. **Clone the Repository:**

   ```
   git clone https://github.com/RajivShah1798/promptly.git

   cd promptly-data-pipeline
   ```

2. **Install Dependencies:**

   ```
   pip install -r requirements.txt

   Setup env file and add SUPABASE_KEY and URL
   ```

3. **Initialize DVC:**

   ```
   dvc init

   dvc remote add gcs_remote gs://promptly-chat

   dvc pull
   ```

4. **Run Airflow DAGs:**

   ```
   airflow db init

   airflow scheduler & airflow webserver

   airflow dags trigger Train_User_Queries
   ```

---

# 6. Evaluation Criteria

The pipeline meets all **required evaluation criteria**:

- **Proper Documentation**
- **Modular Code & Error Handling**
- **DVC Versioning**
- **Pipeline Optimization**
- **Logging and Alerts**