# Assignment 4 - Building a RAG Pipeline with Airflow

Pranjal Mahajan (002375449)

Srushti Patil (002345025)

Ram Srikar Putcha (002304724)

# **Introduction**

In an era where vast amounts of unstructured data are generated daily, efficiently retrieving relevant information from sources like PDFs and web pages is a significant challenge. To address this, our project focuses on building a Retrieval-Augmented Generation (RAG) pipeline using Apache Airflow to automate and optimize data ingestion, processing, and retrieval. By leveraging various PDF parsing strategies, chunking techniques, and vector databases, this system aims to enhance information retrieval and provide precise, context-aware responses.

# **Problem Statement**

Traditional search and retrieval systems struggle with processing and extracting meaningful insights from unstructured data sources. Existing approaches often lack modularity, extensibility, and efficiency when handling large volumes of text, such as NVIDIA’s quarterly reports over the past five years. Additionally, manually computing embeddings and cosine similarity is computationally expensive, limiting scalability. This project aims to overcome these limitations by implementing a robust, automated RAG pipeline that integrates multiple PDF parsing methods, vector databases (Pinecone, ChromaDB), and advanced chunking strategies to optimize retrieval.

# **Project Goals**

Automate Data Processing: Utilize Apache Airflow to orchestrate the end-to-end data pipeline, ensuring efficient ingestion, processing, and retrieval.

Implement PDF Parsing Strategies: Enhance text extraction using Docling, PyMuPDF, and Mistral OCR.

Develop a RAG System:

* Implement manual embedding computation and cosine similarity-based retrieval.
* Integrate with Pinecone and ChromaDB for optimized vector-based retrieval.

Optimize Chunking Strategies: Experiment with section-based, table-based, and sliding window chunking to improve retrieval accuracy.

Build an Interactive UI: Develop a Streamlit-based frontend allowing users to upload PDFs, select parsing and retrieval methods, and query documents dynamically.

Deploy a Scalable System: Containerize the solution using Docker, ensuring modularity and ease of deployment.

# **Proof of Concept**

The project aims to build a Retrieval-Augmented Generation (RAG) pipeline that processes unstructured data from PDF documents, enabling efficient context-aware retrieval and response generation. The PoC validates the feasibility of using Apache Airflow, various PDF parsers, chunking strategies, vector databases, and LLMs to enhance document search and retrieval capabilities.

#### Data Ingestion & Processing

Raw Data Source: NVIDIA’s quarterly reports from the past five years (PDF format).  
Orchestration: Apache Airflow DAG automates data ingestion, processing, and retrieval.  
Storage: Extracted text stored in AWS S3 for efficient access.

#### PDF Parsing Strategies

Docling – For structured PDF parsing.  
PyMuPDF – Lightweight, fast PDF text extraction.  
Mistral OCR – For OCR-based text extraction when dealing with scanned PDFs.

1. Chunking Strategies

Section-Based Chunking – Splits text based on document structure.  
Table-Based Chunking – Extracts tabular data separately.  
Sliding Window Chunking – Ensures context preservation across chunks.

#### Embeddings & Retrieval

RAG Implementation – Computes embeddings & cosine similarity manually.  
Vector Databases:

* Pinecone – Scalable vector search for fast similarity matching.
* ChromaDB – Advanced retrieval with metadata filtering.

Hybrid Search – Allows querying of specific quarters for focused retrieval.

#### Frontend & API Layer

FastAPI Backend – Handles user queries, retrieval, and LLM response generation.  
Streamlit UI –

* Users upload PDFs.
* Select parsing method (Docling, Mistral OCR, etc.).
* Choose retrieval method (Pinecone, ChromaDB).
* Select chunking strategy.
* Query specific time periods.

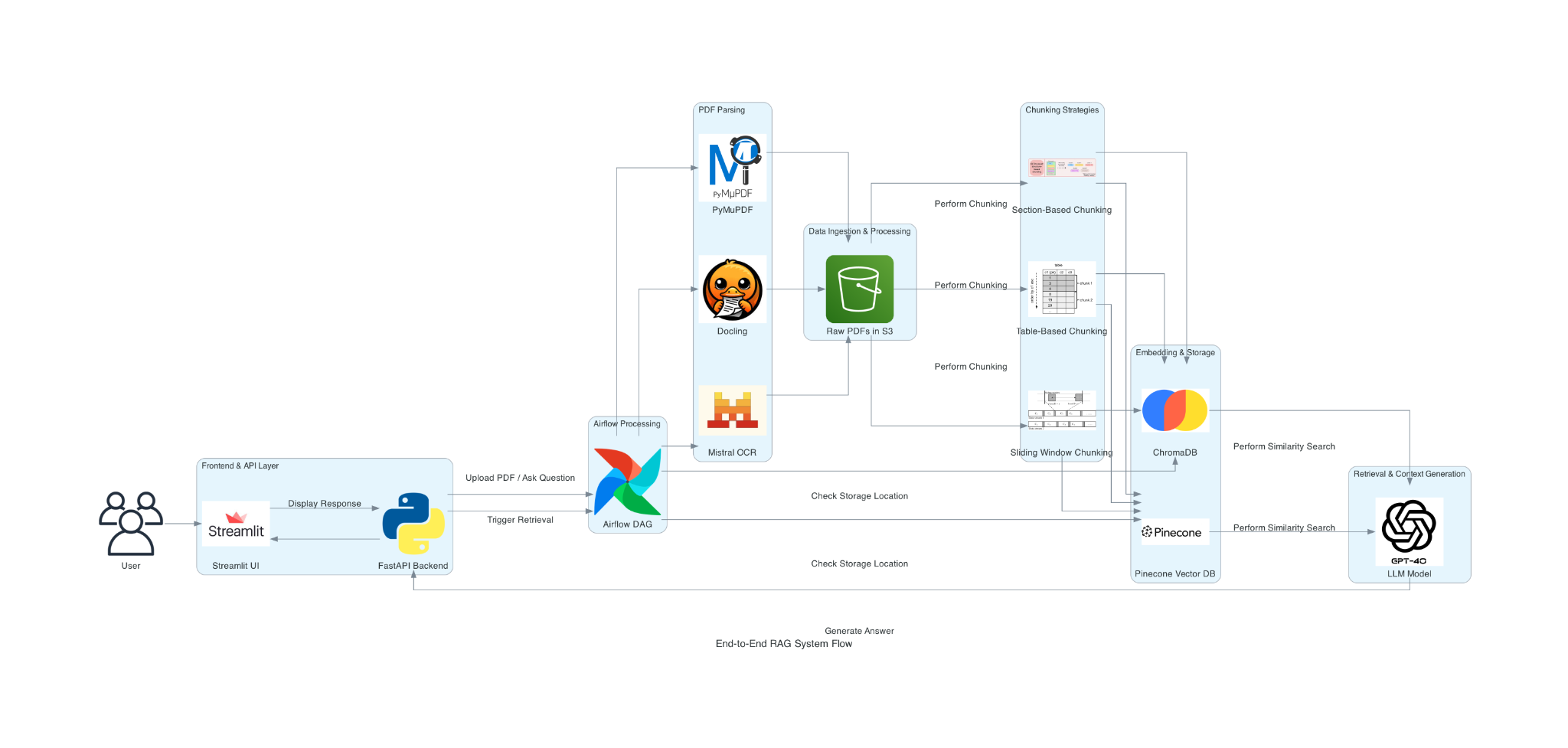
#### LLM Integration & Response Generation

LLM Model (GPT-4.0) – Processes retrieved document chunks to generate relevant, context-aware responses.

The entire system will be containerized using Docker Compose and deployed on the cloud, ensuring scalability and ease of access. This approach will allow for future enhancements, including multi-user support, improved AI model selection, and advanced document processing capabilities.

# **Architecture Diagram**

The architecture diagram represents an End-to-End Retrieval-Augmented Generation (RAG) System Flow for processing and retrieving information from PDF documents.



### 1. User Interaction Layer

Users interact with the system via a web-based UI

* Streamlit UI: Allows users to upload PDFs and ask queries.
* FastAPI Backend: Handles API requests from the UI and communicates with other components.

Users upload a PDF or ask a question related to existing documents.

The request is sent to Airflow DAG for processing.

### 2. Orchestration & Workflow Management

Apache Airflow DAG

* Manages the workflow execution for PDF ingestion, parsing, chunking, embedding, and retrieval.
* Ensures that the pipeline runs smoothly and in sequence.

Triggers retrieval or processing based on user input.

### 3. PDF Parsing Layer

Multiple PDF Parsing Techniques are used to extract text from PDFs:

* PyMuPDF – Fast and efficient PDF text extraction.
* Docling – A library designed for structured document processing.
* Mistral OCR – Optical Character Recognition (OCR) for scanned PDFs.

Extracted text is sent to an AWS S3 bucket (Raw PDFs Storage) for further processing.

### 4. Chunking Strategies

Once PDFs are stored, they are broken down into smaller, retrievable chunks to enhance search efficiency.

* Section-Based Chunking – Splits text based on sections (e.g., headings).
* Table-Based Chunking – Extracts tabular data separately for better retrieval.
* Sliding Window Chunking – Maintains context by overlapping adjacent chunks.

The chunked data is sent for embedding and storage.

### 5. Embedding & Storage

Vector Databases are used to store document embeddings for efficient similarity search:

* ChromaDB – Stores and retrieves document chunks efficiently.
* Pinecone Vector DB – A scalable vector database for high-speed retrieval.

Chunked text is converted into vector embeddings and stored in ChromaDB / Pinecone.

These databases enable fast and accurate similarity searches.

### 6. Retrieval & Context Generation

When a user asks a question, the system performs a similarity search on the vector database.

* ChromaDB / Pinecone returns the most relevant text chunks based on the query.
* The retrieved chunks are fed into an LLM (GPT-4.0) for contextual response generation.

The LLM analyzes the retrieved information and generates an answer.

The response is sent back to FastAPI and displayed in Streamlit UI.

### 7. Response Delivery

* The final answer is presented to the user via Streamlit UI.
* Users get context-aware responses based on retrieved document content.

# **Walkthrough of the Assignment**

## **Step 1: Clone the Repository**

1. Open a terminal or command prompt.

Run the following command to clone the repository:  
git clone <repository\_url>

1. cd Assignment\_4

## **Step 2: Create a Virtual Environment**

Create and activate a virtual environment:  
python -m venv venv

1. source venv/bin/activate # On Windows: venv\Scripts\activate

## **Step 3: Install Dependencies**

1. Install the required dependencies:  
   pip install -r requirements.txt

## **Step 4: Configure Environment Variables**

1. Create a .env file in the root directory.

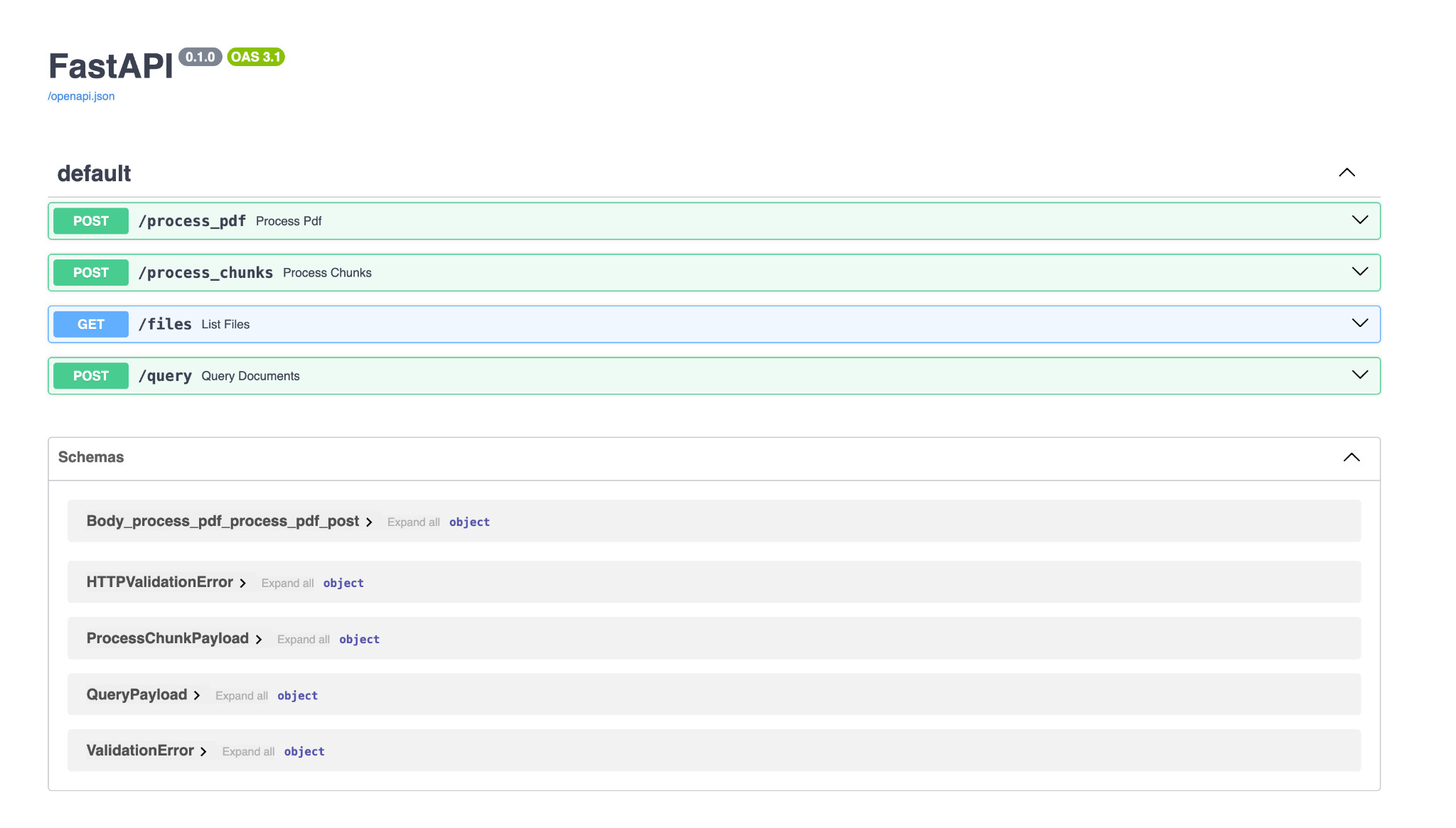
Add required credentials such as API keys for AWS, LLM, etc.

AWS\_ACCESS\_KEY=<your\_access\_key>

AWS\_SECRET\_KEY=<your\_secret\_key>

## **Step 5: Run the Backend Server**

1. Start the FastAPI backend server:  
   uvicorn backend.app:app --reload
2. The API will be available at <http://127.0.0.1:8000>.



## **Step 6: Run the Frontend Dashboard**

1. Start the Streamlit dashboard:  
   streamlit run frontend/dashboard.py
2. Open the displayed local URL to access the dashboard.

## **Step 7: Use the Application**

#### Select a PDF Parser: Choose a PDF parser (Docling, Mistral OCR, PyMuPDF) for text extraction based on the document type.

#### Upload a PDF: Upload an NVIDIA quarterly report, then click "Process" to extract and preprocess the document.

#### Select a RAG Method: Choose a retrieval method: Pinecone vector search ChromaDB for advanced retrieval

#### Choose a Chunking Strategy: Select how the document should be segmented: Section-based chunking Table-based chunking Sliding window chunking

#### Filter by Quarter: Specify the quarterly report data range to refine retrieval accuracy.

#### Ask Questions on the Document: Enter a question related to the document, click "Ask Question," and receive an AI-generated response.

#### Download Processed Data: Select a processed document and click "Download" to retrieve the extracted text and processed insights.

## 

## **Step 8: Verify API Endpoints**

Use Postman or a browser to test API endpoints.

1. Example API request for Uploading a PDF:  
   curl -X POST "http://127.0.0.1:8000/upload\_pdf/" -F "file=@sample.pdf"
2. Example API request for Retrieving Extracted Text from Redis:  
   curl -X GET "http://127.0.0.1:8000/get\_extracted\_text/sample.md"
3. Example API request for Listing Stored Markdown Files in S3:  
   curl -X GET "http://127.0.0.1:8000/select\_pdfcontent/"
4. Example API request for Downloading an Extracted Markdown File:  
   curl -X GET "http://127.0.0.1:8000/download\_markdown/sample.md"
5. Example API request for Summarizing a PDF using an LLM Model:  
   curl -X POST "http://127.0.0.1:8000/summarize/" -F "pdf\_name=sample.md" -F "llm=gpt-4"
6. Example API request for Asking a Question on a PDF using an LLM Model:  
   curl -X POST "http://127.0.0.1:8000/ask\_question/" -F "pdf\_name=sample.md" -F "llm=gemini" -F "question=What are the key points?"
7. Example API request for Fetching the Result of an AI Task (Summarization or Q&A):  
   curl -X GET "http://127.0.0.1:8000/get\_result/task-abc123"

## **Common Issues and Troubleshooting**

1. Chrome WebDriver Not Found: Ensure ChromeDriver is installed, compatible with your Chrome version, and added to the system PATH.
2. Streamlit Dashboard Not Opening: Check for firewall restrictions, manually open the displayed URL, or restart the application.
3. API Requests Failing: Verify that the FastAPI server is running at http://127.0.0.1:8000, and ensure there are no port conflicts.
4. S3 Upload Fails: Confirm AWS credentials (AWS\_ACCESS\_KEY\_ID, AWS\_SECRET\_ACCESS\_KEY) are correctly set in the environment variables and that the bucket exists.
5. PDF Extraction Not Working: Check if the uploaded PDF is valid and ensure PyMuPDF and Azure AI Form Recognizer are installed and configured correctly.
6. Summarization or Q&A Not Processing: Verify that the request is added to Redis (redis-cli MONITOR to check stream data) and that the worker process is running.

#### Chunking Strategies Not Applying: Verify that the selected chunking method is implemented. Debug by printing processed text chunks to confirm expected segmentation.

#### Hybrid Search Not Filtering Properly: Confirm that the selected quarter’s data is correctly indexed and available. Debug query parameters to ensure correct filtering.

1. LLM Model Not Responding: Ensure the selected LLM is available, API keys are valid, and request limits are not exceeded.
2. Airflow DAG Not Executing: Ensure Airflow is running (airflow scheduler & airflow webserver). Check task dependencies and retry failing tasks in the Airflow UI.

# **Application Workflow**

This workflow describes how a user interacts with the system to upload documents, ask questions, and receive responses.

### 1.User Uploads PDF / Asks a Question

* The user uploads a PDF(NVIDIA quarterly reports) via the Streamlit dashboard.
* The user selects PDF parsing method (PyMuPDF, Docking, Mistral OCR)
* The user selects a chunking strategy (section-based, table-based, and sliding window)
* The user selects a database (Pinecone, ChromaDB)
* The user specifies the quarters' data to be used for answering the query.
* The request is submitted to the backend for processing.

### 2. Request Sent (Frontend → Backend - FastAPI)

* Streamlit sends the request to the FastAPI backend via API calls.
* The request contains the PDF file, user query and selected quarter’s data, selected PDF parsing method, chunking strategy and selected RAG method.

### 3. Data Processing (Backend - FastAPI & Airflow)

Document Processing:

* If the PDF is new, (PyMuPDF, Docling, Mistral OCR) extracts the text for processing.
* The extracted text is stored in AWS S3 for future access.

#### Chunking & Retrieval:

* The backend applies the selected chunking strategy (e.g., fixed-size, semantic-based).
* Hybrid search allows the system to retrieve only the relevant quarter’s data.
* The processed data is indexed in Pinecone/ChromaDB for optimized retrieval.

#### Query Processing & LLM Response

* The backend retrieves relevant document chunks based on the query.
* The processed data is sent to an LLM model for response generation.
* The LLM model generates a structured response.

### 4. Data Storage (AWS S3 & Vector Databases)

* Extracted text, embeddings, and processed responses are stored in AWS S3.
* Pinecone/ChromaDB stores vector representations for efficient search and retrieval.

### 5. Data Retrieval (API Layer - FastAPI)

* If the same document is uploaded again, pre-processed data is fetched from AWS S3.
* The backend provides API endpoints for retrieving processed data and LLM responses.

### 6. Visualization (Frontend - Streamlit Dashboard)

* The frontend fetches and displays the AI-generated response.
* The response includes extracted document content, retrieved relevant chunks, LLM-generated answer
* Users can compare different retrieval methods and chunking strategies by modifying parameters.

### 7. Deployment (Docker & Airflow Pipelines)

* Airflow Pipeline: Manages data ingestion, processing, and retrieval.
* Streamlit + FastAPI Pipeline: Handles user interaction and query processing.

# **References**

* [**Docling GitHub Repository**](https://github.com/docling-project/docling)
* [**Mistral OCR Documentation**](https://docs.mistral.ai/capabilities/document/)
* [**Apache Airflow Documentation**](https://airflow.apache.org/docs/)
* [**Pinecone Documentation**](https://docs.pinecone.io/guides/get-started/overview)
* [**ChromaDB Documentation**](https://docs.trychroma.com/docs/overview/introduction)
* **FastAPI & Streamlit Documentation**
* [**AWS S3 Best Practices**](https://aws.amazon.com/s3/)

# **Disclosures**

WE ATTEST THAT WE HAVEN’T USED ANY OTHER STUDENTS’ WORK IN OUR

ASSIGNMENT AND ABIDE BY THE POLICIES LISTED IN THE STUDENT HANDBOOK

| Pranjal Mahajan(002375449) | 33.33% |
| --- | --- |
| Srushti Patil (002345025) | 33.33% |
| Ram Putcha (002304724 | 33.33% |

AiUseDisclousure :

We have utilized AI to assist us in the following areas:

* Understanding how to deploy FastAPI for rendering and the necessary deployment steps.
* Deploying a Streamlit frontend application to Streamlit Community Cloud.
* Connecting frontend and backend deployments to work seamlessly together
* Learning how to implement RAG pipelines
* Understanding how Pinecone and ChromaDB stores data and accessing it