# **End-to-End ML + LLM Pipeline Orchestration**

This project presents a robust, full-cycle orchestration pipeline designed for both traditional Machine Learning (ML) models and modern Retrieval-Augmented Generation (RAG) style Large Language Model (LLM) systems. It demonstrates automated data processing, model training, versioning, and scalable deployment by leveraging a powerful suite of open-source MLOps tools.

## **Features at a Glance**

This project offers a comprehensive set of features that streamline the development and deployment of both traditional ML and advanced LLM applications:

* **Complete MLOps Cycle:** The pipeline encompasses the entire Machine Learning Operations (MLOps) lifecycle, from raw data ingestion and processing through model training, versioning, and ultimately, deployment for real-time inference. This end-to-end automation significantly reduces manual effort and ensures consistency across all stages.
* **Hybrid AI Approach:** A key strength of this project is its ability to seamlessly integrate two distinct AI paradigms: a classical ML model for predictive tasks and a cutting-edge RAG-style LLM for contextualized natural language understanding and generation. This hybrid approach allows for a broader range of AI solutions within a unified orchestration framework.
* **Scalable & Efficient:** For handling large datasets and computationally intensive tasks, the project utilizes **Apache Spark** (specifically PySpark) for distributed data processing. This ensures that the pipelines can scale efficiently to meet the demands of real-world big data scenarios.
* **Reproducible ML:** **MLflow** plays a crucial role in ensuring the reproducibility of machine learning experiments. It provides comprehensive tracking of model parameters, metrics, and artifacts, as well as robust versioning and registration capabilities. This allows developers to easily revert to previous model versions, compare experiments, and maintain a clear lineage of all trained models.
* **Production-Ready Deployment:** The project leverages **FastAPI** to power the web services for real-time inference. FastAPI is known for its high performance and ease of use, making it an ideal choice for deploying ML and LLM models as robust and scalable APIs.
* **Containerized Environments:** To guarantee consistent and isolated setups across different environments (development, testing, production), **Docker and Docker Compose** are extensively used. This containerization strategy eliminates "it works on my machine" issues and simplifies deployment by bundling all dependencies and configurations into portable containers.
* **Smart LLM Interactions:** The RAG system implemented in this project enables highly contextualized question answering from custom documents. By retrieving relevant information from a knowledge base before generating a response, the LLM can provide more accurate, informed, and hallucination-free answers.

## **Core Technologies**

The project is built upon a foundation of leading open-source technologies, each serving a critical role in the overall pipeline:

| Category | Technology | Description |
| --- | --- | --- |
| **Orchestration** | **Apache Airflow** | Serves as the central workflow management system, responsible for defining, scheduling, and monitoring directed acyclic graphs (DAGs) that orchestrate the entire ML pipeline. |
| **Data Processing** | **Apache Spark (PySpark)** | A powerful distributed processing engine used for handling large datasets, performing complex data transformations, and executing feature engineering steps with high efficiency. |
| **MLOps** | **MLflow** | Provides a platform for managing the ML lifecycle, including tracking experiments, logging parameters and metrics, registering and versioning models, and facilitating model deployment. |
| **Web Services** | **FastAPI** | A modern, fast (high-performance) web framework for building APIs. It is used to expose the trained ML and RAG LLM models for real-time inference. |
| **Vector DB** | **FAISS** | (Facebook AI Similarity Search) An open-source library for efficient similarity search and clustering of dense vectors. It is crucial for the RAG system's ability to quickly retrieve relevant document chunks. |
| **LLMs & Embeddings** | **Hugging Face transformers & sentence-transformers** | Provides access to a vast collection of pre-trained models for various language tasks (e.g., text generation, summarization) and efficient methods for generating dense vector embeddings from text. |
| **Containerization** | **Docker & Docker Compose** | Tools for packaging applications and their dependencies into portable containers, ensuring consistent environments and simplifying the deployment and scaling of services. |

Export to Sheets

## **Project Structure**

The project is organized into a modular and intuitive directory structure, making it easy to navigate and understand the different components:

ML\_LLM\_PIPELINE/

├── airflow/ # Airflow DAGs for ML pipeline orchestration

├── api/ # FastAPI app for ML model inference

├── data/ # Shared directory for ML input data (e.g., WineQuality.csv)

├── models/ # Stores trained ML models (e.g., model.pkl)

├── mlruns/ # MLflow tracking server data

├── rag/ # Part B: RAG-style LLM Pipeline components

│ ├── ingest.py # Script for document ingestion, chunking, embedding, and FAISS index creation

│ ├── query.py # Core RAG logic: retrieve relevant chunks, generate answer with LLM

│ ├── app.py # FastAPI server for the RAG endpoint

│ ├── vectorstore/ # Stores FAISS index and embeddings

│ ├── data/ # Stores raw input documents for RAG (e.g., PDFs, CSVs)

│ ├── Dockerfile # Dockerfile for building the RAG application image

│ ├── requirements.txt # Python dependencies for the RAG application

│ └── docker-compose.yml # Docker Compose for orchestrating the RAG service

├── scripts/ # PySpark training script for ML pipeline

├── docker-compose.yml # Main Docker Compose for the entire ML pipeline (Part A)

└── README.md # This file (You are here!)

* **airflow/**: Contains the Apache Airflow DAGs responsible for orchestrating the end-to-end ML pipeline. These DAGs define the sequence of tasks, dependencies, and schedules for data processing, model training, and registration.
* **api/**: Houses the FastAPI application that serves the trained ML model for real-time inference. This directory typically contains the API endpoint definitions and the logic for loading and making predictions with the model.
* **data/**: A shared directory intended for storing input data for the traditional ML pipeline. For example, WineQuality.csv would reside here.
* **models/**: This directory is used to store trained machine learning models, often in formats like .pkl or other serialization formats, before they are potentially registered with MLflow.
* **mlruns/**: This directory is automatically generated by MLflow and stores all tracking data for experiments, including parameters, metrics, artifacts, and run information.
* **rag/**: This is a dedicated sub-directory for the Retrieval-Augmented Generation (RAG) style LLM pipeline (Part B). It is self-contained and includes:
  + ingest.py: A script for processing raw documents (e.g., PDFs), chunking them into smaller pieces, generating embeddings for these chunks, and building a FAISS index for efficient similarity search.
  + query.py: Contains the core logic for the RAG system, handling the retrieval of relevant document chunks based on a user query and then using an LLM to generate a coherent answer.
  + app.py: The FastAPI server specifically for the RAG endpoint, allowing external applications to query the RAG system.
  + vectorstore/: Stores the generated FAISS index and embeddings, which are critical for the RAG system's performance.
  + data/: Stores the raw input documents (e.g., PDFs, CSVs) that the RAG system will use as its knowledge base.
  + Dockerfile: Defines the steps to build a Docker image for the RAG application, encapsulating all its dependencies.
  + requirements.txt: Lists the Python dependencies required for the RAG application.
  + docker-compose.yml: A Docker Compose file specifically for orchestrating the RAG service, allowing it to be run independently.
* **scripts/**: This directory holds the PySpark training script for the ML pipeline. This script contains the logic for data loading, preprocessing, model training, and potentially MLflow logging.
* **docker-compose.yml**: The main Docker Compose file at the root of the project. It defines and orchestrates all the services for the entire ML pipeline (Part A), including Airflow, MLflow, Spark, and the ML API.
* **README.md**: The main documentation file for the project.

## **Part A: Wine Quality Prediction (ML Pipeline)**

This section details the automated, end-to-end machine learning workflow designed for predicting wine quality. It highlights the seamless integration of various MLOps tools to ensure a robust and reproducible ML lifecycle.

### **Key Features:**

The Wine Quality Prediction ML Pipeline is built with several core features to ensure efficiency, scalability, and reproducibility:

* **Automated Training DAG:** At the heart of this pipeline is an **Apache Airflow DAG (Directed Acyclic Graph)**. This DAG orchestrates the entire ML training process. From the initial data preparation and feature engineering steps to the actual model training and subsequent registration, Airflow automates the workflow, making it reliable and repeatable. This eliminates manual interventions and ensures that every step is executed in the correct sequence.
* **Scalable Data Transformation:** To handle potentially large datasets efficiently, the pipeline leverages **PySpark** for data preprocessing and feature engineering. PySpark's distributed computing capabilities allow for scalable data transformations, ensuring that the pipeline can process significant volumes of data quickly and effectively, which is crucial for real-world ML applications.
* **MLflow Integration:** **MLflow** is deeply integrated into this pipeline to ensure meticulous tracking, versioning, and registration of machine learning models. Every experiment, including its parameters, metrics, and artifacts, is logged in MLflow. This provides a clear audit trail and enables easy comparison of different model runs. The MLflow Model Registry is utilized to manage different versions of models and promote "production" models, ensuring that only validated models are deployed.
* **Production Model Serving:** The pipeline is designed for seamless deployment. The latest production-ready model, as designated in the MLflow Model Registry, is automatically loaded and served via a dedicated **FastAPI** endpoint. This setup enables real-time inference, allowing other applications to consume predictions from the trained ML model with low latency.
* **Containerized Environment:** The entire ML pipeline, encompassing Airflow, Spark, MLflow, and FastAPI, is encapsulated within **Docker containers**. This containerization strategy guarantees a consistent and isolated environment across all stages of development and deployment. It eliminates dependency conflicts and simplifies the setup process, ensuring that the pipeline behaves identically regardless of the underlying infrastructure.

### **How to Run Part A:**

Follow these steps to set up and run the Wine Quality Prediction ML pipeline:

1. **Ensure Docker Desktop is running.** All services are containerized, so Docker Desktop must be active on your machine.
2. **Navigate to the root directory** of the project: ML\_LLM\_Pipeline/. This is where the main docker-compose.yml file is located.

**Start all services** defined in the main docker-compose.yml. This command will build the necessary Docker images (if not already built) and bring up the Airflow, MLflow, Spark, and ML API services:  
Bash  
docker-compose up --build

1. This process might take some time on the first run as it downloads base images and installs dependencies.
2. **Access Airflow UI:** Once all services are up and running, open your web browser and navigate to the Airflow user interface: [http://localhost:8080](https://www.google.com/search?q=http://localhost:8080).
   * You might need to log in to Airflow. The default credentials are typically airflow for both username and password.
3. **Unpause and Trigger DAG:** In the Airflow UI, locate the ML pipeline DAG. It should be named something like wine\_quality\_prediction\_dag.
   * First, **unpause** the DAG by toggling its status.
   * Then, manually **trigger** the DAG. This action will initiate the entire workflow, including data processing, model training, and model registration in MLflow. You can monitor the progress of the tasks within the Airflow UI.
4. **Access MLflow UI (Optional):** To monitor the experiments, tracked runs, and registered models, you can access the MLflow UI at: [http://localhost:5000](https://www.google.com/search?q=http://localhost:5000). Here, you can see details about each model training run, compare metrics, and manage model versions.
5. **Access ML Prediction API:** Once the ML pipeline has successfully completed and a model is registered and served, you can interact with the FastAPI documentation for the ML model's prediction endpoint: [http://localhost:8000/docs](https://www.google.com/search?q=http://localhost:8000/docs). This interface allows you to send test requests to the deployed model and receive predictions.

## **Part B: RAG-style LLM Pipeline (Mini POC)**

This section outlines the implementation of a Retrieval-Augmented Generation (RAG) system, showcasing how a small, open-source Large Language Model (LLM) can answer questions based on custom, ingested documents. This mini Proof of Concept (POC) demonstrates a practical application of LLMs for domain-specific knowledge retrieval.

### **Key Features:**

The RAG-style LLM Pipeline is designed with several critical features to facilitate contextualized and accurate question answering:

* **Document Ingestion:** The pipeline is equipped to load and process various document types, including common formats like PDF and CSV. This flexibility allows users to integrate their proprietary or domain-specific documents into the RAG system's knowledge base. The ingestion process typically involves reading the document content and preparing it for further processing.
* **Semantic Search:** To enable intelligent retrieval of relevant information, the system utilizes **sentence-transformers** for generating robust semantic embeddings from text. These embeddings capture the meaning of the text, allowing for more accurate similarity comparisons. **FAISS (Facebook AI Similarity Search)** is employed as a high-performance vector database. FAISS is optimized for efficient similarity search over large collections of dense vectors, which is crucial for quickly identifying and retrieving the most relevant document chunks based on a user's query.
* **Contextualized Generation:** A compact, open-source LLM (e.g., google/flan-t5-small) forms the core of the generation component. This LLM is **augmented with retrieved context** from the FAISS vector store. Instead of relying solely on its pre-trained knowledge, the LLM is provided with specific, relevant information extracted from your custom documents. This augmentation significantly enhances the LLM's ability to provide accurate, relevant, and grounded answers to natural language queries, reducing the likelihood of hallucinations.
* **FastAPI Q&A Interface:** For easy interaction, a dedicated **FastAPI endpoint (/rag-query/)** is provided. This endpoint offers a clean and efficient REST interface, allowing other applications or users to send natural language questions to the RAG system and receive generated answers. FastAPI's high performance ensures responsive interactions.
* **Self-Contained Deployment:** The entire RAG component is designed for modularity and independent deployment. It is **independently containerized** using its own Dockerfile and docker-compose.yml located within the rag/ directory. This self-contained nature simplifies deployment, makes it easy to integrate the RAG service into larger architectures, and ensures consistency across different environments.

### **How to Run Part B:**

Follow these steps to set up and interact with the RAG-style LLM pipeline:

**Navigate to the RAG directory:** Open your terminal and change your current directory to the rag subdirectory within the main project:  
Bash  
cd ML\_LLM\_Pipeline/rag

1. **Place your data:** Ensure that the PDF documents or other supported data files you want the RAG system to learn from are located in the ML\_LLM\_Pipeline/rag/data/ directory. For example, if you have a PDF named Global warming.pdf, place it here.

**Run Ingestion (First time & whenever data changes):** This is a crucial preprocessing step. It reads your documents, splits them into manageable chunks, generates vector embeddings for each chunk, and builds the FAISS index. This index is then used for efficient semantic search. These generated artifacts will be mounted into the Docker container when the service runs.  
Bash  
python ingest.py

1. Upon successful execution, you should see confirmation messages similar to: Indexed X chunks from: rag/data/Global warming.pdf. **Note:** You must run this command on your host machine, not inside a Docker container, as it prepares the data for the container.

**Build the Docker Image:** This command reads the Dockerfile and requirements.txt within the rag/ directory to create the Docker image for your RAG application.  
Bash  
docker-compose build rag-app

1. **Note:** This step may take a considerable amount of time, especially on the first run, as it involves downloading base images, installing Python libraries, and potentially downloading pre-trained LLM model weights. Ensure you have a stable internet connection.

**Start the RAG Service:** This command will start the FastAPI application inside a Docker container, making the RAG service accessible.  
Bash  
docker-compose up rag-app

1. The application will be accessible via [http://localhost:8002](https://www.google.com/search?q=http://localhost:8002). You will see output from Uvicorn in your terminal, indicating that the FastAPI server is running and processing requests. This command will keep running and attach to the container's logs.
2. **Test the RAG API:** While the docker-compose up rag-app command is still running in its terminal window, open a new terminal window or your web browser to test the API endpoints:

**Health Check:** To verify that the RAG API is running, access:  
http://localhost:8002/

* + Expected output: {"message": "API running"}

**Query RAG System:** To send a query to the RAG system, use the /rag-query/ endpoint. Replace "What is global warming?" with a question relevant to the content of your ingested PDF documents:  
http://localhost:8002/rag-query/?question=What is global warming?

* + You should receive a JSON response containing the original question, the retrieved context (the relevant chunks from your documents), and the LLM's generated answer based on that context.

1. **Stop the RAG Service:** To stop the running RAG service, go back to the terminal where you executed docker-compose up rag-app and press Ctrl+C. This will gracefully shut down the Docker container.

## **Contributing**

Contributions to this project are highly encouraged and welcome! If you have suggestions for improvements, want to add new features, or have identified any bugs, please feel free to contribute. You can do so by:

* **Opening an Issue:** If you find a bug, have a feature request, or want to propose a significant change, please open a new issue on the GitHub repository. Provide a clear and detailed description of your suggestion or the problem you've encountered.
* **Submitting a Pull Request:** If you have implemented a fix or a new feature, you can submit a pull request. Please ensure your code adheres to the project's style and includes appropriate tests if applicable.

Your contributions help make this project better for everyone!

## **License**

This project is licensed under the **MIT License**. The MIT License is a permissive free software license that allows you to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the software, provided that the original copyright and license notice are included in all copies or substantial portions of the software.

For the full details of the license, please refer to the [LICENSE](https://www.google.com/search?q=LICENSE) file in the project's root directory.

## **Acknowledgments**

This project stands on the shoulders of giants, and we extend our sincere gratitude to:

* **The open-source community** for generously providing these incredible tools, libraries, and frameworks (Apache Airflow, Apache Spark, MLflow, FastAPI, Hugging Face, FAISS, Docker, etc.) that make complex MLOps pipelines achievable.
* **Your inspiration** for making this README more beautiful and comprehensive!