**LANGCHAIN-BASED ASSISTANT FOR MOVIE KNOWLEDGE RETRIEVAL**

**INTRODUCTION:**

In the evolving landscape of intelligent systems, the ability to extract meaningful insights from unstructured and semi-structured data is becoming essential. This project bridges the gap between raw movie metadata and human-like conversational interfaces by transforming complex data into actionable, conversational insights.

Built upon the sample\_mflix dataset from MongoDB Atlas, the system performs a comprehensive transformation of NoSQL collections into a fully normalized SQL relational schema. It then applies cutting-edge Natural Language Processing (NLP) techniques to facilitate seamless interaction via natural language queries.

To support diverse use cases and performance preferences, the system offers dual vector storage backends:

* **FAISS**, for efficient local retrieval, and
* **Pinecone**, for scalable cloud-based retrieval- allowing flexibility based on deployment needs

For embedding generation, **Hugging Face models** are utilized to convert textual data into high-dimensional vectors. The response generation is powered by **Groq-based LLM inference**, ensuring ultra-fast, context-aware answers. The architecture is built using **LangChain’s Retrieval-Augmented Generation (RAG)** framework. While the current setup uses Hugging Face and Groq, it is fully compatible with **OpenAI embeddings and models** for future extensibility.

This end-to-end system showcases expertise in data engineering, vectorization, retriever-augmented pipelines, and LLM integration. The final solution is deployed through a clean, interactive **Streamlit interface**, delivering intelligent responses in real-time.

**PROBLEM STATEMENT:**

With the exponential rise of multimedia content and user engagement on streaming platforms, retrieving relevant and contextually accurate information from large-scale movie databases has become increasingly complex. Traditional keyword-based search mechanisms often fall short, particularly when data is dispersed across unstructured formats such as JSON documents in NoSQL systems.

This project tackles that challenge by converting unstructured movie metadata into a structured, intelligent, and searchable question-answering system. Leveraging the sample\_mflix dataset from MongoDB, the solution involves transforming NoSQL collections into a normalized relational SQL database, followed by the generation of semantic embeddings using **Hugging Face models**.

A **Retrieval-Augmented Generation (RAG)** architecture lies at the core of the system, enabling it to deliver fluent, human-like answers to natural language queries. The pipeline supports both **FAISS (local)** and **Pinecone (cloud)** vector stores for adaptable retrieval, while **Groq-based LLMs** power real-time response generation. The system is also compatible with **OpenAI models and embeddings** for future extensibility.

The final product is a fast, context-aware movie Q&A system deployed via a **Streamlit interface**, offering a smooth and interactive user experience.

**OBJECTIVES:**

The primary objective of this project is to develop an intelligent movie-based question answering system by combining structured SQL data and unstructured natural language processing techniques using a Retrieval-Augmented Generation (RAG) architecture.

The specific goals include:

1. **Migrate Data from MongoDB to SQL:**  
   Transform unstructured JSON documents from the sample\_mflix MongoDB dataset into a fully normalized SQL relational format. Establish foreign key relationships to organize data across entities like movies, users, comments, sessions, and theaters.
2. **Design SQL Schema & Model Relationships:**  
   Create a robust relational schema to manage entities like movies, users, comments, sessions, and theaters efficiently.
3. **Preprocess and Embed Textual Content:**  
   Merge relevant textual fields—such as movie plots, cast, genres, and reviews—into cohesive documents. Generate semantic vector embeddings using Hugging Face models, with optional compatibility for OpenAI embeddings.
4. **Build Vector Store with FAISS and Pinecone (Dual Support):**  
   Enable flexible vector indexing by offering both FAISS (for local storage and retrieval) and Pinecone (for cloud-based retrieval), based on deployment or scalability needs.
5. **Implement RAG Architecture via LangChain:**  
   Use LangChain's Retrieval-Augmented Generation (RAG) pipeline to retrieve semantically relevant movie information and generate natural, context-aware answers using Groq-based large language models, with fallback compatibility for OpenAI LLMs.
6. **Develop a Streamlit-Based QA Interface:**  
   Create a clean, intuitive interface using Streamlit, allowing users to ask natural language questions and receive real-time, intelligent movie-related responses
7. **Evaluate System Performance:**  
   Assess the system on key performance metrics such as response accuracy, semantic relevance, user interface usability, and retrieval latency under both FAISS and Pinecone configurations.

**DATASET DESCRIPTION:**

The dataset used in this project is the **sample\_mflix** dataset, a publicly available movie dataset hosted on **MongoDB Atlas**. It simulates the backend of a movie streaming platform, containing comprehensive data across multiple collections related to movies, users, comments, and session details.

**Collections Used:**

* **movies** – Contains metadata such as title, plot, genre, cast, directors, languages, runtime, and release information.
* **comments** – User-submitted movie reviews linked to specific movies.
* **embedded\_movies** – Additional movie entries embedded within documents.
* **users** – User profiles, including login and identification data.
* **sessions** – Session details to simulate login behavior and activity.
* **theaters** – Geolocation and details of theaters, useful for linking screening locations.

**Dataset Characteristics:**

* **Source Format:** JSON documents (MongoDB)
* **Target Format:** Relational Tables (MySQL)
* **Total Collections Extracted:** 6
* **Size:** Medium-scale, real-world simulation
* **Domain Coverage:** Movies, reviews, user interactions, streaming platform behavior

**Preprocessing Done:**

* Flattened nested fields (e.g., cast, genres)
* Converted arrays into comma-separated strings
* Renamed columns to match SQL schema
* Ensured data types were compatible with relational storage

**ENVIRONMENT CONFIGURATION (.env FILE):**

To ensure security, flexibility, and cleaner code management, all sensitive credentials and configuration variables are stored in a .env file. This file is loaded using the python-dotenv library and is excluded from version control to protect private information.

**Environment Variables Used**

| **Variable Name** | **Description** |
| --- | --- |
| mongodb\_uri | MongoDB Atlas connection string for extracting collections |
| db\_user | Username for the SQL (MySQL/PostgreSQL) database |
| db\_password | Password for the SQL database |
| db\_host | Host address for the SQL server (e.g., localhost or remote endpoint) |
| db\_name | Name of the target SQL database |
| api\_key\_1, api\_key\_2, api\_key\_3, api\_key\_4, api\_key\_5 | Rotating **OpenAI API keys** for embedding generation and LLM calls (if used) |
| groq\_api\_key | API key for accessing **Groq LLM inference** |
| pinecone\_api\_key | API key for accessing **Pinecone vector store** |
| pinecone\_env | Pinecone environment (e.g., gcp-starter, us-west1-gcp) |
| pinecone\_index\_name | Name of the active index used in Pinecone |

**METHODOLOGY:**

The project follows a modular pipeline involving data migration, transformation, semantic embedding, and intelligent response generation using RAG architecture. The complete workflow is divided into the following phases:

**1. Data Extraction & Transformation**

* Extracted collections (movies, comments, users, sessions, theaters, embedded\_movies) from **MongoDB Atlas** using custom Python ETL scripts with **pymongo**.
* Flattened nested fields such as cast, genres, and languages into comma-separated strings for SQL compatibility.
* Transformed the extracted JSON documents into cleaned **Pandas DataFrames**, with proper field renaming, typecasting.
* Designed a **relational SQL schema**
* Loaded the structured data into a **MySQL** database.

**2. Preprocessing & Embedding Generation**

* Combined key text fields like title, plot, reviews, and cast into cohesive movie-level documents.
* Removed short or irrelevant entries and normalized text (case, symbols, etc.).
* Generated **semantic vector embeddings** using **Hugging Face models** (with fallback compatibility for **OpenAI embeddings** via api\_key\_1 to api\_key\_5).
* Stored the resulting vectors in either one of them based on the requirement:
  + **FAISS** (for fast local similarity search)
  + **Pinecone** (for scalable cloud-based retrieval)

**3. RAG Architecture Implementation**

* Integrated **LangChain** to implement a flexible **Retrieval-Augmented Generation (RAG)** pipeline.
* Dynamically switched between **FAISS** and **Pinecone** retrievers based on configuration.
* Connected the retriever output to a **Groq-powered LLM** for ultra-fast generation of human-like responses, with optional support for **OpenAI GPT models**.
* Ensured real-time performance with high semantic relevance between queries and retrieved movie metadata.

**4. QA Interface Development**

* Developed an interactive and clean **Streamlit interface** for user interaction.
* Implemented an input box for natural language questions and an output display for AI-generated responses.
* Integrated backend logic to:
  + Retrieve Embeddings
  + Fetch relevant context using RAG
  + Generate answers using the selected LLM
* Ensured responsive layout, minimal latency, and clear output formatting to deliver a smooth and engaging user experience.

**EVALUATION METRICS:**

The system’s performance and effectiveness were assessed based on the following key metrics:

| **Metric** | **Description** |
| --- | --- |
| **Data migration completeness** | All collections from MongoDB were properly mapped, cleaned, and pushed into the SQL database. |
| **Embedding accuracy** | Ensures that semantically similar queries return contextually similar results from the vector store. |
| **Response relevance** | Evaluated based on human judgment for how precise and relevant the generated answers are to user queries. |
| **System latency** | Measures the time taken from the user’s prompt submission to the delivery of the final response. |
| **Usability of the interface** | Assessed based on user feedback regarding the ease of use, clarity, and responsiveness of the QA interface. |

**TECHNICAL STACK & TOOLS USED:**

| **Category** | **Tools/Technologies** |
| --- | --- |
| **Database** | MongoDB (NoSQL), MySQL |
| **Data Migration** | Python (pymongo, SQLAlchemy, Pandas) |
| **Data Transformation** | Python (Data cleaning, flattening, schema mapping) |
| **Embeddings** | OpenAI Embedding API |
| **Vector Store** | FAISS (Facebook AI Similarity Search), Pinecone (cloud-based vector store) |
| **RAG Framework** | LangChain |
| **Language Model** | OpenAI GPT (LLM for generating responses), Groq |
| **Frontend UI** | Streamlit (Web-based QA interface) |
| **Environment Tools** | .env, dotenv, Git, |

**RELEVENT DOCUMENT LINKS:**

**Python Libraries**

| **Library/Tool** | **Documentation Link** |
| --- | --- |
| **PyMongo** | <https://pymongo.readthedocs.io/en/stable/> |
| **python-dotenv** | <https://saurabh-kumar.com/python-dotenv/> |
| **pandas** | <https://pandas.pydata.org/docs/> |
| **SQLAlchemy** | <https://docs.sqlalchemy.org/en/20/> |
| **mysql-connector-python** | <https://dev.mysql.com/doc/connector-python/en/> |
| **openai** | <https://platform.openai.com/docs/libraries/python-sdk> |
| **langchain** | <https://python.langchain.com/docs/> |
| **langchain-community** | [https://python.langchain.com/docs/modules/model\_io/llms/integrations/](https://www.google.com/search?q=https://python.langchain.com/docs/modules/model_io/llms/integrations/) |
| **tiktoken** | <https://github.com/openai/tiktoken> |
| **numpy** | <https://numpy.org/doc/> |
| **streamlit** | <https://docs.streamlit.io/> |
| **faiss-cpu** | <https://github.com/facebookresearch/faiss/wiki/> (Main Faiss project, CPU specific documentation is usually within this) |
| **fpdf** | <https://pyfpdf.readthedocs.io/en/latest/> |
| **langchain-groq** | [https://python.langchain.com/docs/integrations/llms/groq](https://www.google.com/search?q=https://python.langchain.com/docs/integrations/llms/groq) |
| **sentence-transformers** | <https://www.sbert.net/docs/index.html> |
| **langchain-huggingface** | [https://python.langchain.com/docs/integrations/text\_embedding/huggingface\_embeddings](https://www.google.com/search?q=https://python.langchain.com/docs/integrations/text_embedding/huggingface_embeddings) |
| **PyTorch (torch)** | <https://pytorch.org/docs/stable/index.html> |
| **pinecone-client** | <https://docs.pinecone.io/docs/> |
| **langchain-pinecone** | <https://python.langchain.com/docs/integrations/vectorstores/pinecone> |
| **langchain-openai** | <https://python.langchain.com/docs/integrations/llms/openai> |

**Specific Document Links**

| **Library/Tool** | **Documentation Link** |
| --- | --- |
| **MongoDB Documentation** | <https://www.mongodb.com/docs/> |
| **Pinecone Documentation** | <https://docs.pinecone.io/> |
| **Docker Documentation** | <https://docs.docker.com/> |
| **Docker Hub Documentation** | <https://docs.docker.com/docker-hub/> |
| **Amazon EC2 Documentation** | <https://docs.aws.amazon.com/ec2/> |

**MONGODB SETUP GUIDE:**

* Go to <https://www.mongodb.com/>
* Sign IN using **Google**
* Select **Create Cluster** and create a cluster
* In **Services** select **Atlas Search** and select **Collections Tab**
* Search for sample\_mflix database
* Go to **clusters**
* Select **Connect**
* Configure **IP** address
* Configure **Driver** and **Version**
* Select **Done**

"Once the **connection is established**, you can fetch data from MongoDB using **PyMongo** in Python."

**MONGODB SETUP GUIDE:**

* Go to <https://www.pinecone.io/>
* Sign IN into pinecone
* Select **Generate API Key** and store the API key in .env file

**DOCKERIZE YOUR STREAMLIT APP GUIDE:**

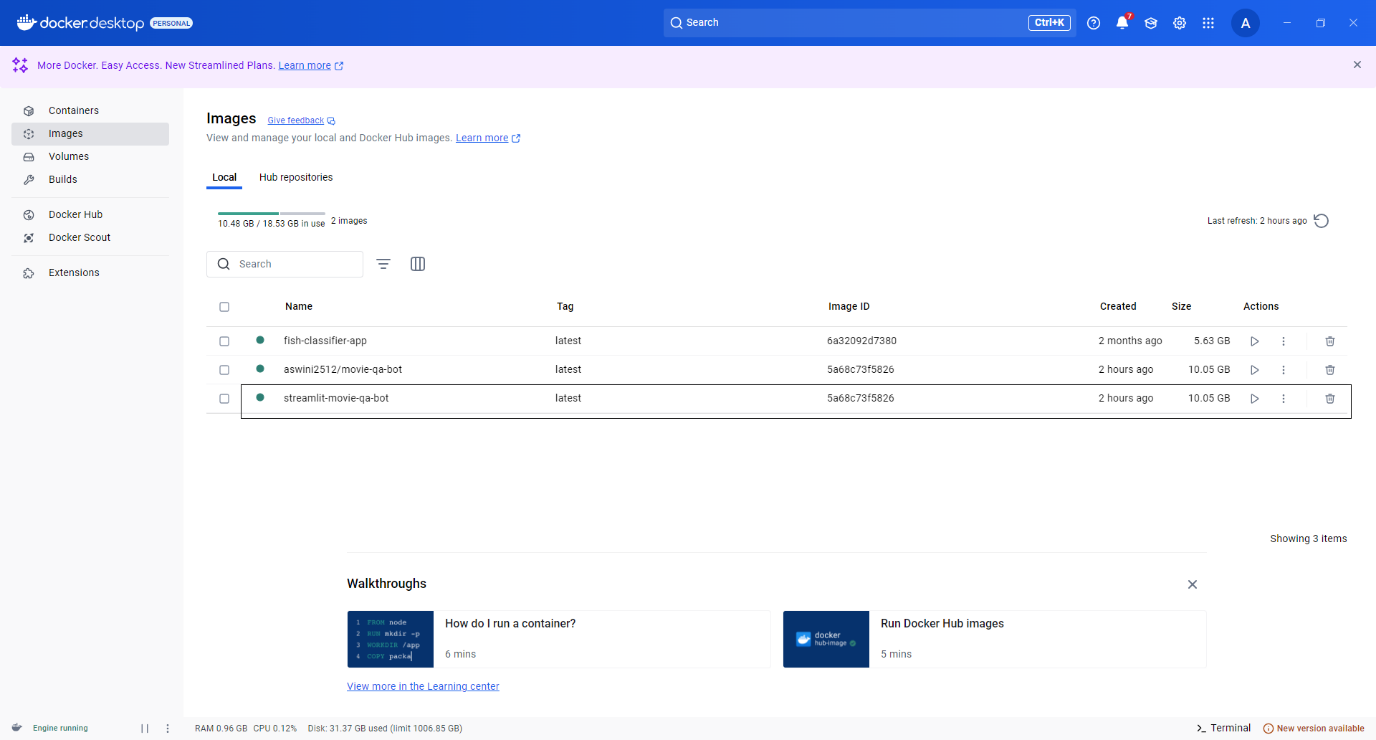
* Install docker desktop in your local
* Create the following essential files
  + Dockerfile
  + .dockerignore
  + Requirements.txt

Do this inside your **project root directory** (where chatbot.py or main.py lives).

* Build your docker image
  + docker build -t streamlit-movie-qa .

here, streamlit-movie-qa is just a tag name (you can name it anything). It creates an image from your project

* Run Container locally:
  + docker run -p 8501:8501 streamlit-movie-qa

****

Just like in the image, the **Docker image** you've built will be visible in your **Docker Desktop** interface.

**DOCKER TO EC2 DEPLOYMENT GUIDE:**

**Step 1: Sign Up & Create Repository on Docker Hub**

* Go to <https://hub.docker.com/>
* Create an account
* After signing in:
  + Click **Repositories** 🡪 **Create Repository**
  + Provide Name 🡪 ex “movie-qa-bot”
  + Visibility: Public/Private
  + Click **Create**

**Step 2: Tag & Push Image to Docker Hub (from local)**

* Tag the docker image

Docker tag <image\_name> <dockerhub\_username>/<repository\_name>:<tag\_name>

* docker tag: This is the command used to create a new tag for an existing Docker image.
* <image\_name>: This represents the **name of the Docker image you have already built locally**.
* <dockerhub\_username>: This is your **Docker Hub username**. This is crucial if you intend to push this image to your Docker Hub repository later.
* <repository\_name>: This is the **name you want to give to the image within your Docker Hub repository**. It often matches the local image name or is similar. In your example, this was movie-qa-bot.
* <tag\_name>: This is the **specific tag or version identifier** for your image. This is optional, but highly recommended for version control. If you omit it, Docker defaults to latest.
* Login to docker hub

docker login

Enter Docker Hub username and password.

* Push the Docker image

docker push <dockerhub\_username>/<repository\_name>:<tag\_name>

**Step 3: Create & set up EC2 Instance**

* Launch new instance
* AMI: Amazon Linux 2
* Instance type: t3.micro
* Key pair: Create .pem and download
* Storage: Set at least **20-30** **GB**
* Security Group:
* Allow **SSH (port 22)** from 0.0.0.0/0
* Allow **Custom TCP (port 8501)** from 0.0.0.0/0

**Step 4: SSH into EC2 from Local Machine**

In Git Bash

ssh -i <path\_to\_your\_.pem\_key\_file> ec2-user@<public\_ip \_of\_ec2\_instance>

**Step 5: Install Docker on EC2**

* sudo yum update -y
* sudo yum install docker -y
* sudo service docker start
* sudo usermod -a -G docker ec2-user
* exit

reconnect after adding docker permissions:

ssh -i <path\_to\_your\_.pem\_key\_file> ec2-user@<public\_ip \_of\_ec2\_instance>

**Step 6: Install Docker on EC2**

* Login inside EC2:

docker login

* Pull the image

docker pull <dockerhub\_username>/<repository\_name>:<tag\_name>

**Step 7: Transfer .env File from Local to EC2**

* From local Git Bash:

scp -i <.pem file name> .env ec2-user@<public-ip>:~

This uploads the .env to EC2 home directory.

**Step 8: Run the Docker Container**

* From local Git Bash:

docker run --env-file .env -p 8501:8501 < username>/<repository\_name>:<tag\_name>

* Streamlit app now runs on - http://<public-ip>:8501

**Step 9: Stop container**

* docker ps # Get container ID
* docker stop <container\_id>

**CONCLUSION:**

This project successfully demonstrates the integration of data engineering and natural language processing to build an intelligent, real-time movie question-answering system. Starting from unstructured MongoDB collections, the data was transformed into a structured SQL format, semantically vectorized, and queried using a Retrieval-Augmented Generation (RAG) approach with LangChain and OpenAI.

The system effectively retrieves relevant movie information and responds to natural language queries through an interactive interface. This end-to-end solution not only showcases the practical use of LLMs in media applications but also lays the foundation for more scalable AI-powered assistants across domains.