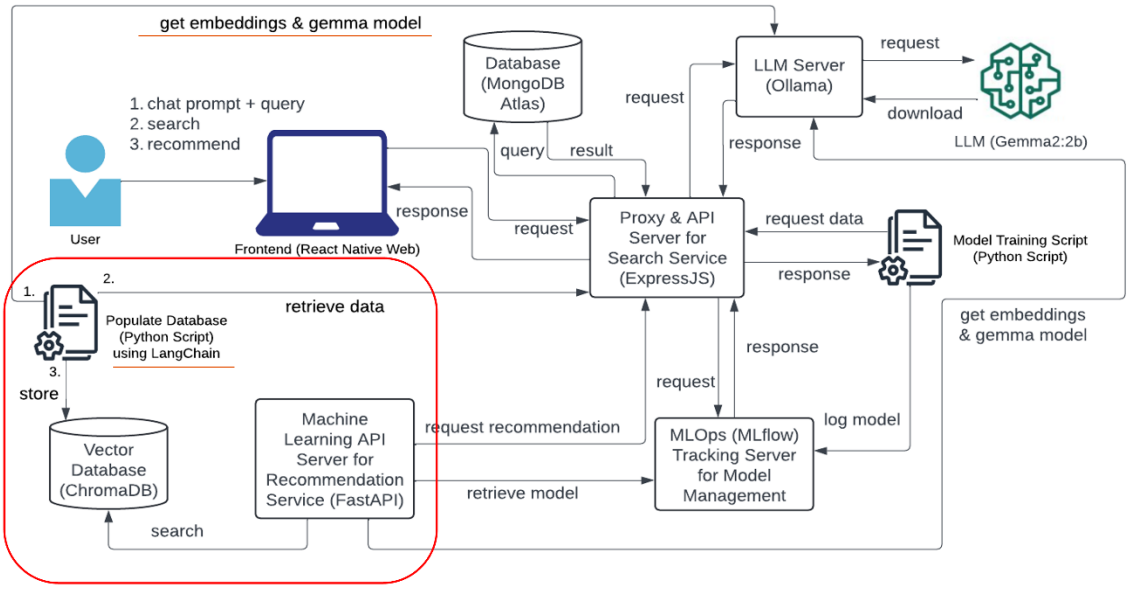
**DIT 637 Smart and Secure Systems**
**TT09A Experiencing RAG through a LLM Framework and Vector DB**
08/17/2024 Developed by Clark Ngo
08/17/2024 Reviewed by Sam Chung
School of Technology & Computing (STC) @ City University of Seattle (CityU)

## References


- Schwaber-Cohen, R. (n.d.). *What is a Vector Database & How Does it Work? Use Cases + Examples*. Pinecone. https://www.pinecone.io/learn/vector-database/
- Chroma. An open-source AI application database (2024). Trychroma.com. https://www.trychroma.com/
- *LangChain*. (n.d.). https://www.langchain.com/


## Key Concepts and Tools for Experiencing RAG




The diagram highlight the key components involved in the **model training and deployment** process. Here's a breakdown of their functions:

**Populate Database (Python Script) using LangChain:**

- This component is responsible for fetching the necessary data from the database (MongoDB Atlas) and inserting it to a vector database (Chroma DB).

**Get Embeddings & Gemma Model:**

- This part involves transforming the data into a numerical representation (embeddings) that the model can understand.
- The Gemma model also helps with embedding generation.

**LangChain – LLM application framework**

LangChain is a powerful framework designed to simplify the creation of applications using large language models (LLMs). It provides a flexible and modular approach to building applications that leverage the capabilities of LLMs

**Chroma DB - Vector Databases**

**Chroma DB** is an open-source embedding database designed for large language models (LLMs) and other AI applications. Chroma is co-founded by Jeff Huber and Anton Troynikov.

In the diagram, the **Vector DB (Chroma DB)** is a specialized database designed to store, manage, and retrieve high-dimensional vector data, often called embeddings. These embeddings are numerical representations of data objects that capture their semantic meaning and context, such as text, images, or other unstructured data types.

Features of a Vector DB

- Storage of Vectors: Vector databases store data as vectors, arrays of numbers representing various attributes or features of the data.
- Similarity Search: They enable efficient similarity search operations, allowing you to find data points close to each other in the vector space.
- High-dimensional Data Handling: These databases are optimized for handling high-dimensional data, which traditional databases struggle with.

Why Use a Vector Database?

- Efficient Retrieval: Vector databases provide low-latency queries, making them ideal for real-time applications that require fast and accurate data retrieval.
- Scalability: They are designed to scale horizontally, efficiently handling large volumes of data and high query loads.
- Enhanced AI Applications: By storing embeddings generated by AI models, vector databases enable advanced AI applications like semantic search, recommendation systems, and more.
- Versatility: They can handle various data types, including text, images, and audio, making them versatile for different use cases.

Use Cases

- Recommendation Systems: Finding similar items based on user preferences.
- Image and Video Search: Searching for visually similar images or videos.
- Natural Language Processing: Enhancing search engines with semantic understanding.
- Anomaly Detection: Identifying unusual patterns in data for security or maintenance purposes.

Other Examples

- Pinecone: A managed vector database service known for its high performance and scalability.
- Milvus: An open-source vector database that excels in high-speed vector similarity search and scalability.

- Weaviate: This open-source vector search engine handles various data types, including text and images. It offers semantic search capabilities and integrates well with machine learning models.
- Faiss: Developed by Facebook AI Research, Faiss is a library for efficient similarity search and clustering of dense vectors. It's particularly useful for large-scale data.
- Qdrant: An open-source vector database focusing on high-performance and reliable vector search capabilities. It's suitable for various machine learning and AI applications.

**Ollama - A lightweight, extensible framework designed to run LLMs on local machines**
In the diagram, the **LLM Server (Ollama)** section handles requests related to the large language model (LLM) operations, particularly for the chat interface in the frontend. Jeffrey Morgan and Michael Chiang developed Ollama. It stands for Omni-layer Learning Language Acquisition Model.

Ollama is used for its advanced ability to understand and generate human-like text, enabling various applications from automation to educational support. It can run a variety of models, including Llama, Mistral, and Gemma. Using advanced language models like those from OpenAI can streamline research by automating literature reviews, enhancing data analysis, and aiding in writing complex academic papers.

1. **Frontend Interaction:**
   - The user interacts with the chat interface in the React Native web frontend, sending prompts or queries.
   - These prompts are sent as requests to the **LLM Server (Ollama)** via the **Proxy & API Server (ExpressJS)**.
2. **LLM Server (Ollama):**
   - The **LLM Server (Ollama)** receives and processes these requests using a third-party language model, such as a Generative Pre-trained Transformer (GPT).
   - The **LLM Server** might either directly request the third-party LLM/GPT service or use a locally downloaded model to generate responses.
3. **Streaming Response:**
   - The **LLM Server** is configured to send streaming responses back to the frontend. This means that as the LLM generates a response, it starts sending it back immediately rather than waiting for the entire response to be ready. It creates a smoother, real-time chat experience for the user.
4. **Communication Flow:**
   - The processed response from the **LLM Server (Ollama)** is then sent back to the **Proxy & API Server (ExpressJS)**.
   - The **ExpressJS** server forwards the response to the React Native frontend, which is displayed in the chat interface.
5. **Third-party LLM/GPT:**
   - The third-party LLM/GPT (outlined in the red box) could be a service like OpenAI's GPT-3 or another similar model. It is the engine that generates the natural language responses based on the user's input.

   - **Complexity in ExpressJS:** The service might become complex and more challenging to maintain as it takes on more responsibilities.

**LLM with Gemma Model**
Gemma is a family of lightweight, advanced open models developed by Google DeepMind, building on the research and technology behind the Gemini models. Named after the Latin word for "precious stone," Gemma models are designed for easy integration into applications, mobile devices, or hosted services. They can be customized using tuning techniques to excel at specific tasks. Supported by developer tools that encourage innovation and responsible AI use, Gemma models are intended for the AI development community to extend and enhance.

Gemma 2 and Gemma are key models in the Gemma family, designed for generative AI tasks like text generation, question answering, summarization, and reasoning. Built on the same research as the Gemini models, these base models come in various sizes, ranging from **2 billion to 27 billion parameters**, and can be tuned for specific use cases.

**What are the parameters in the context of machine learning and deep learning?**
Parameters refer to the components within a model learned from the training data. These parameters are essential for the model's ability to make predictions or generate outputs based on new inputs.

- **Weights and Biases:** In neural networks, parameters typically include weights and biases. Weights are the values that determine the strength of the connections between neurons in different layers of the network, while biases are additional constants added to the inputs of a layer to shift the activation function.
- **Training Process:** During training, the model adjusts these parameters to minimize the difference between its predictions and the actual target values (the loss). This process is typically done using optimization algorithms like gradient descent.
- **Model Size:** The number of parameters in a model often corresponds to its complexity and capacity to learn intricate patterns. For example, larger models with billions of parameters can capture more detailed relationships in the data, but they also require more computational resources and may need larger datasets to train effectively.

**Inputs and outputs**
**Input:** A text string, such as a question, a prompt, or a document to be summarized.
**Output:** Generated English-language text in response to the input, such as an answer to a question or a summary of a document.

**Model Data**
Data used for model training and how the data was processed.

**Training Dataset**
These models were trained on a dataset of text data that includes a wide variety of sources. The 27B model was trained with 13 trillion tokens, the 9B model was trained with 8 trillion tokens, and the 2B model was trained with 2 trillion tokens. Here are the key components:

- Web Documents: A diverse collection of web text ensures the model is exposed to a broad range of linguistic styles, topics, and vocabulary. Primarily English-language content.
- Code: Exposing the model to code helps it learn the syntax and patterns of programming languages, which improves its ability to generate code or understand code-related questions.

- Mathematics: Training on mathematical text helps the model learn logical reasoning and symbolic representation and address mathematical queries.

The combination of these diverse data sources is crucial for training a powerful language model that can handle a wide variety of different tasks and text formats.

**Data Preprocessing**
Here are the key data cleaning and filtering methods applied to the training data:

- CSAM Filtering: Rigorous CSAM (Child Sexual Abuse Material) filtering was applied at multiple stages in the data preparation process to ensure the exclusion of harmful and illegal content.
- Sensitive Data Filtering: As part of making Gemma pre-trained models safe and reliable, automated techniques were used to filter out certain personal information and other sensitive data from training sets.
- Additional methods: Filtering based on content quality and safety in line with our policies.

**Benefits**
At the time of release, this family of models provides high-performance, open large language model implementations designed from the ground up for Responsible AI development compared to similarly sized models.

Using the benchmark evaluation metrics described in this document, these models have shown to provide superior performance to other, comparably sized open model alternatives.

**ExpressJS as Proxy Server and API Gateway**
To streamline frontend requests to various services, we restructured our architecture so that ExpressJS serves as the sole service endpoint. ExpressJS will forward requests to different services, including the database, machine learning API server, MLOps tracking server (MLflow), and LLM server (Ollama). As with any system design, it is essential to consider and discuss the pros and cons of the architecture or design.

**Pros:**

- **Simplified Frontend:** A single endpoint (ExpressJS) reduces complexity in the frontend, making it easier to manage.
- **Centralized Routing:** ExpressJS handles routing, providing a single control point for directing traffic to the appropriate services.
- **Scalability:** The architecture can scale by managing different services behind the scenes, allowing independent scaling of backend components.
- **Security:** It is easier to enforce security measures like authentication and rate limiting at a single-entry point.

**Cons:**

- **Single Point of Failure:** If ExpressJS fails, the entire system can be affected.
- **Increased Latency:** Routing requests through ExpressJS adds an extra layer, potentially increasing response times.
- **Bottleneck Risk:** High traffic could overwhelm ExpressJS, causing a bottleneck if not properly scaled.

***Uploading three image files to your GitHub Repository generated from GitHub Classroom***

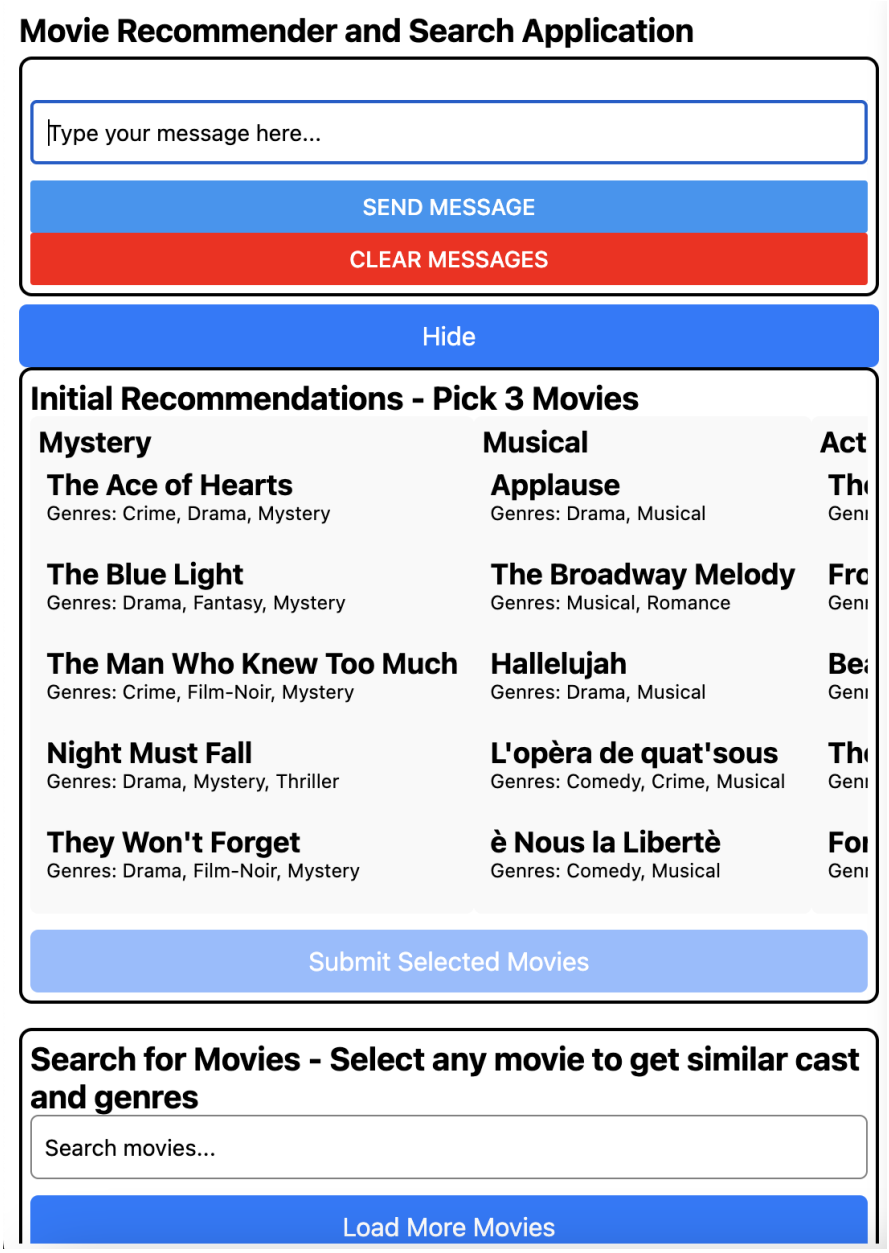
1. The screenshot of your 'LOAD DATA TO CHROMADB' as 'firstname_lastname_load_data_to_chromadb.png.'
2. The screenshot of your 'MOVIE NO INFO' as 'firstname_lastname_movie_no_info.png.'
3. The screenshot of your 'MOVIE HAS INFO' as 'firstname_lastname_movie_has_info.png.'

### 1) Use Case

As a movie enthusiast using a mobile device, I want to search and browse a list of movies with details such as title, genre, and year to quickly find information about movies I am interested in while on the go.

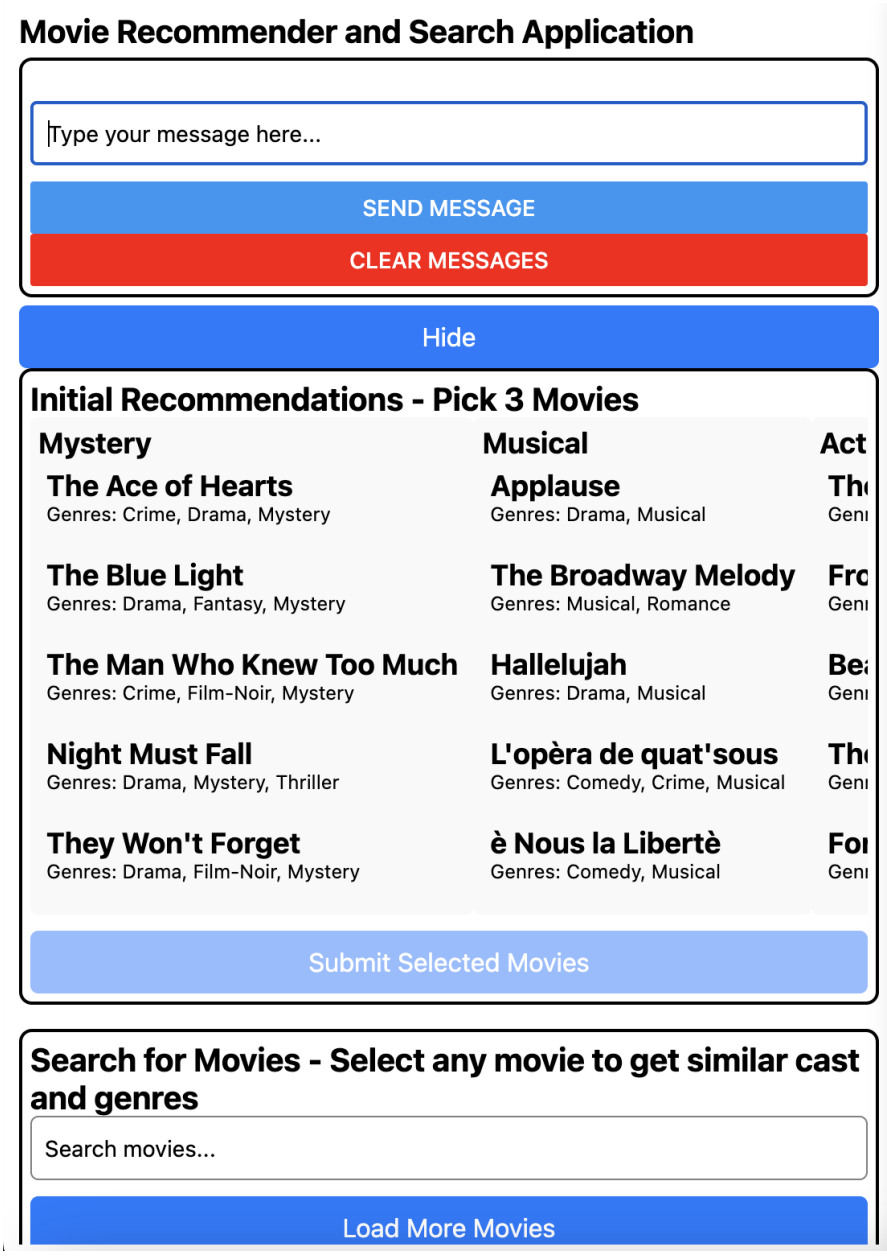
As a movie enthusiast using a mobile device, I want to get movie recommendations from my selections.

As a movie enthusiast using a mobile device, I want to chat and get movie recommendations from my selections.



Features included:

- Chat Recommendation (RAG)
- Movie List Display
- Search Functionality
- Initial Recommendations
- New Recommendations
- Similar Cast and Genres

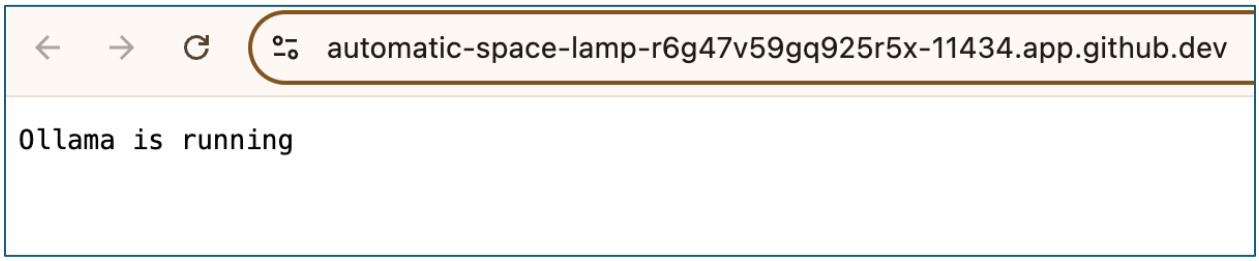


## 2) Setup

### *2.1) Running the Ollama LLM Server*

- Create/Open GitHub Codespaces.
- Open a terminal and type the following:
  - **curl -fsSL https://ollama.com/install.sh | sh**
    The command curl -fsSL https://ollama.com/install.sh | sh is used to download and execute a shell script from the specified URL. This specific script is designed to install Ollama on a Linux system. It detects the operating system architecture and installs the appropriate version of Ollama. The script also handles setup tasks, such as creating necessary directories and setting permissions.
    - curl: This is a command-line tool for transferring data with URLs.
    - -f: This option tells curl to fail silently when there are server errors.
    - -s: This makes curl operate silently without showing progress or error messages.
    - -S: When used with -s, it makes curl show an error message if it fails.
    - -L: This option tells curl to follow redirects.
    - https://ollama.com/install.sh: This is the URL of the script you want to download.
    - | sh: This pipes the downloaded script to the sh command, which executes it.
  - **ollama serve**
    This command will start the server, and you can then interact with it using the Ollama API or other tools that communicate with the server.

- Test the Forwarded Address in the Browser. You can find the URL (port number: 11434) from the “PORTS” tab.



- Open a terminal and type the following:
  - **ollama pull gemma2:2b**
    This command will download the Gemma 2 model with 2 billion parameters to your local environment, making it ready for use.
  - **ollama run gemma2:2b**
    This command will start the Gemma 2 model, making it ready for inference tasks.

**Gemma** is a family of lightweight, state-of-the-art open models developed by Google. These models are designed for responsible AI development and are built from the same research and technology used to create the Gemini models

```
@clarkjasonngo → /workspaces/DIT637-TT08 (main) $ ollama run gemma2:2b
>>> limit 50 words. recommend me sci-fi movies.
**For mind-bending thrills:**

* **Inception (2010):** Dream sequences, elaborate heists, and mind-blowing twists.
* **The Matrix (1999):** Hacking reality and questioning the truth with spectacular action.


**For awe-inspiring adventures:**

* **Interstellar (2014):** Space exploration and human connection across galaxies.
* **Arrival (2016):** Communicating with alien life forms in a thrilling mystery.



Enjoy!  🚀


>>> Send a message (/? for help)
```

Another Example

```
@samchung0117 →/workspaces/DIT637-TT09 (main) $ ollama pull gemma2:2b
pulling manifest
pulling 7462734796d6... 100% ▕████████████████████████████████████▏ 1.6 GB
pulling e0a42594d802... 100% ▕████████████████████████████████████▏  358 B
pulling 097a36493f71... 100% ▕████████████████████████████████████▏ 8.4 KB
pulling 2490e7468436... 100% ▕████████████████████████████████████▏   65 B
pulling e18ad7af7efb... 100% ▕████████████████████████████████████▏  487 B
verifying sha256 digest
writing manifest
removing any unused layers
success
@samchung0117 →/workspaces/DIT637-TT09 (main) $ ollama run gemma2:2b
>>> Recommend me 3 sci-fi. Limit 25 words.
1. **Dune** (Epic space opera with political intrigue and a harsh desert world)
2. **The Martian** (Survival story of an astronaut stranded on Mars)
3. **The Hitchhiker's Guide to the Galaxy** (Humorous, thought-provoking exploration of space)


>>> Send a message (/? for help)
```

- To exit the chat, press:
  - CTRL + D

### *2.2) Running the Backend ExpressJS Application Server*

- Create a **.env** file inside 'backend-expressjs' folder (use the 'example.env' file as a reference)
  - Update the MONGODB_URI= connection string
- Open a new terminal:
  - **cd backend-expressjs**
  - **npm install**
  - **npm run start**
- Make the Port Public
- Copy the Forwarded Address (URL): silver-waffle-v6pxvxv7gwxfpr79-3000.app.github.dev
- Do a quick test if its running

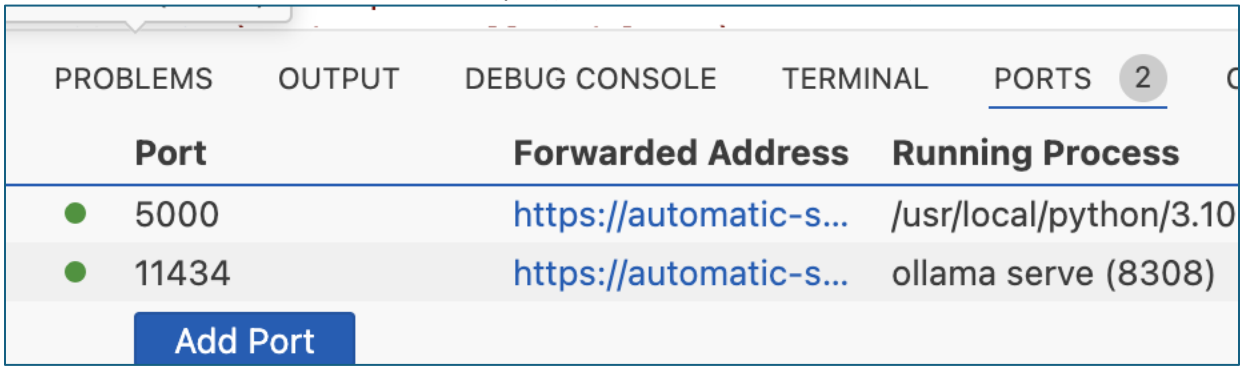
### *2.3) Running the MLflow Tracking Server*

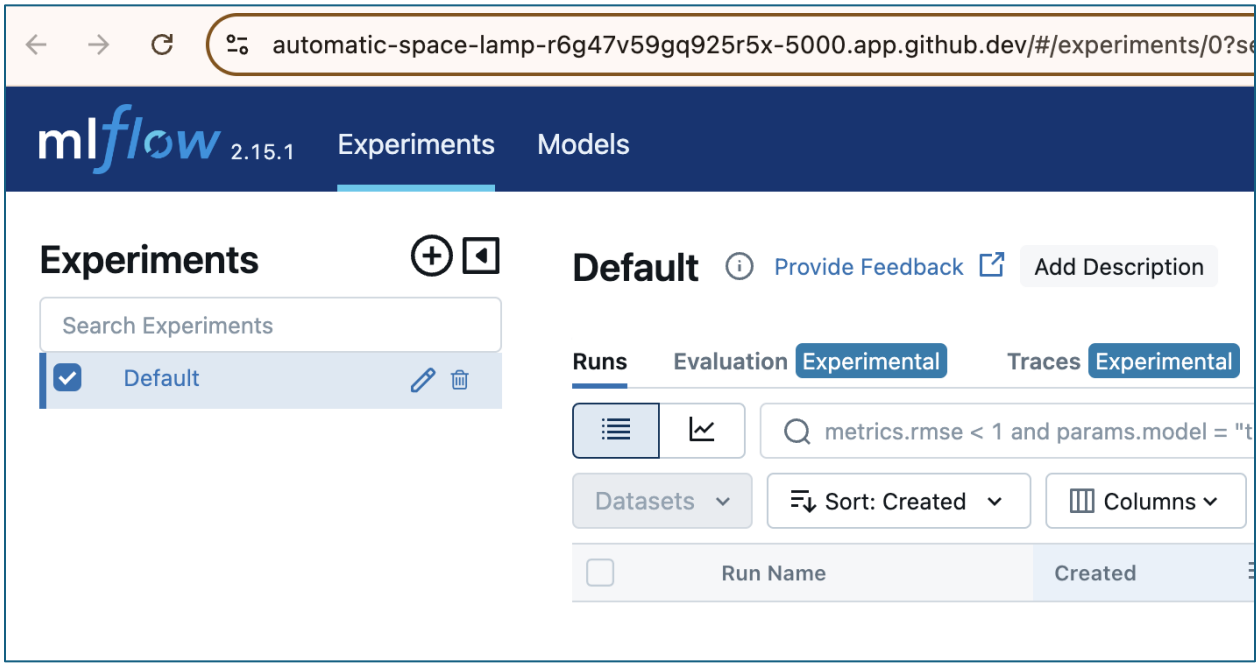
- Open a terminal and type the following:
  - **cd mlops_mlflow**
  - **pip install -r requirements.txt**
    pip reads the "requirements.txt" file and installs all the listed packages with the specified versions
  - **mlflow server**
- Test the Forwarded Address in the Browser

```
@clarkjasonngo → /workspaces/DIT637-TT08/mlops-mlflow (main) $ mlflow server
[2024-08-17 17:41:56 +0000] [14512] [INFO] Starting gunicorn 22.0.0
[2024-08-17 17:41:56 +0000] [14512] [INFO] Listening at: http://127.0.0.1:5000 (14512)
[2024-08-17 17:41:56 +0000] [14512] [INFO] Using worker: sync
[2024-08-17 17:41:56 +0000] [14513] [INFO] Booting worker with pid: 14513
[2024-08-17 17:41:56 +0000] [14514] [INFO] Booting worker with pid: 14514
[2024-08-17 17:41:57 +0000] [14515] [INFO] Booting worker with pid: 14515
[2024-08-17 17:41:57 +0000] [14516] [INFO] Booting worker with pid: 14516
```

*Access the MLflow Tracking Server*

- Click the 127.0.0.1:5000 in the output (or access this in the Ports tab)



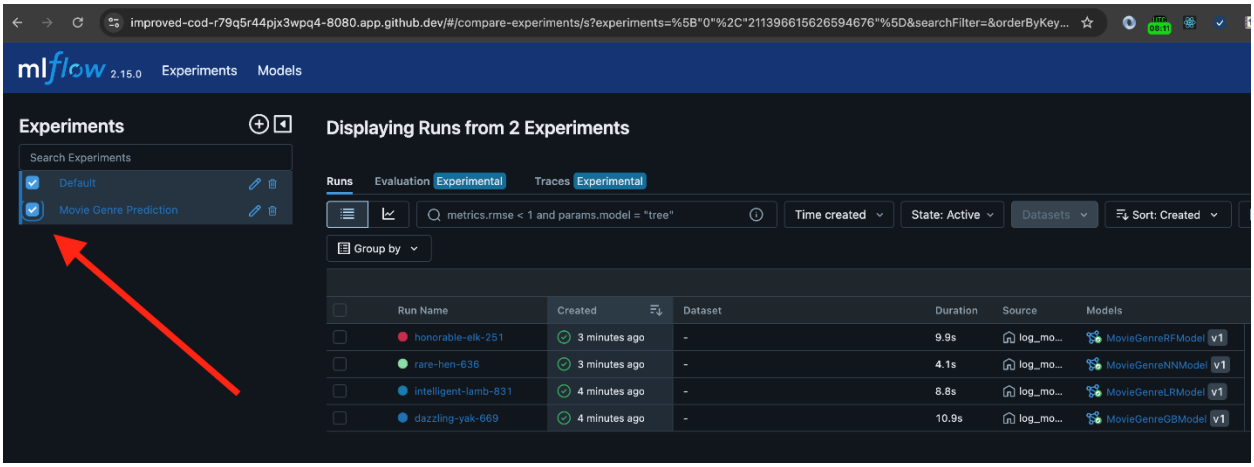
### *2.4) Running the Python Scripts for ML Model Deployment (under the 'mlops-mlflow' directory)*

*Train different model and prepare metadata for logging*

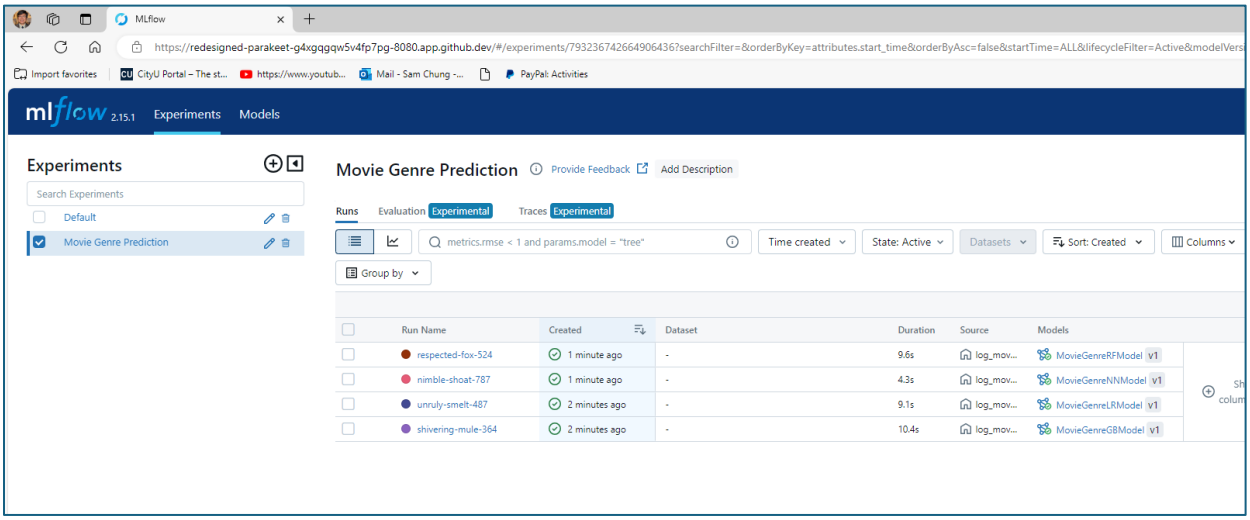
- Open a new terminal:
  - **cd mlops-mlflow**
  - **python run_all_models.py**
- New folders 'mlartifacts' and 'mlruns' and their respective files will be generated and will be used as data in the MLflow Tracking Server.
  - **ML Artifacts:** Data files, models, and code created during machine learning experiments, stored for reproducibility, tracking, and reuse in future projects.
  - **ML Runs:** Individual executions of machine learning experiments, including configurations, parameters, and results, tracked to compare and improve models.


*View the Experiments in the MLflow Tracking Server*

- Head back to the server by clicking the Forwarded Address in the Ports tab.
- Tick the checkbox for Movie Genre Prediction to view all 4 models that were run.

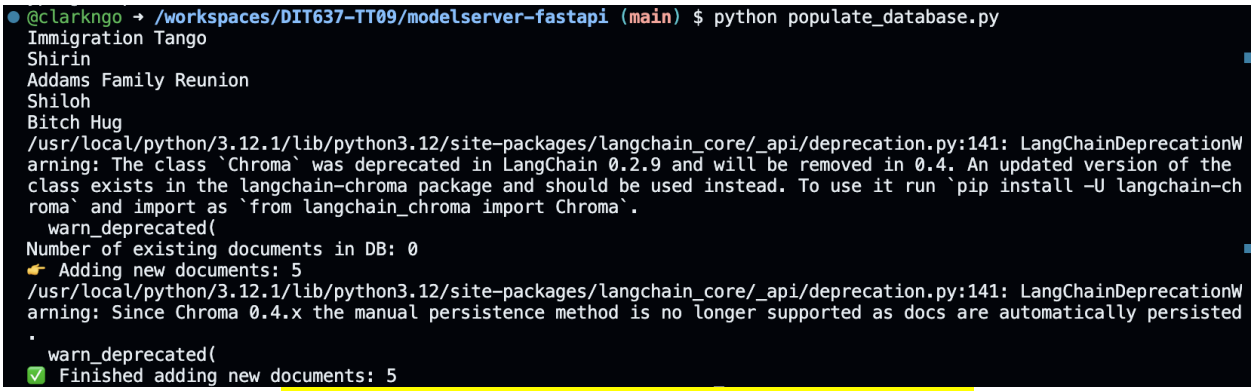


For example,


MLflow
mlflow 2.15.1
Experiments
Models
Experiments
Search Experiments
Default
Movie Genre Prediction
Movie Genre Prediction
Provide Feedback
Add Description
Runs
Evaluation Experimental
Traces Experimental
metrics.rmse < 1 and params.model = "tree"
Time created
State: Active
Datasets
Sort: Created
Columns
Group by
Run Name
Created
Dataset
Duration
Source
Models
respected-fox-524
1 minute ago
9.6s
MovieGenreRFModel v1
nimble-shoat-787
1 minute ago
4.3s
MovieGenreNNModel v1
unruly-smelt-487
2 minutes ago
9.1s
MovieGenreLRModel v1
shivering-mule-364
2 minutes ago
10.4s
MovieGenreGBModel v1

### *2.5) Running the Python Script to Create ChromaDB to Insert Data and Running the Machine Learning Model FastAPI Server*

- Open a new terminal:
  - **cd modelserver-fastapi**
  - **pip install –r requirements.txt**
  - **python populate_database.py**




Take a screenshot of your 'LOAD DATA TO CHROMADB' as 'firstname_lastname_load_data_to_chromadb.png'.

As this is a lightweight application, we will only load 5 random movies in the Chroma DB vector database. For example:

- Immigration Tango
- Shirin
- Addams Family Reunion
- Shiloh
- Bitch Hug

The Happiest Millionaire
Dot the I
Caribe
Kira kira hikaru

Test for Hallucinations
Test 1: Movie that is listed should return text related to that movie

- In the same terminal, type the following:
  - **python query_data.py "What is <any_movie_title_listed> movie?"**
  - **Example:**
    - **python query_data.py "What is Addams Family?"**

```
@clarkngo → /workspaces/DIT637-TT09/modelserver-fastapi (main) $ python query_data.py "What is Addams Family?"
/usr/local/python/3.12.1/lib/python3.12/site-packages/langchain_core/_api/deprecation.py:141: LangChainDeprecationW
arning: The class `Chroma` was deprecated in LangChain 0.2.9 and will be removed in 0.4. An updated version of the
class exists in the langchain-chroma package and should be used instead. To use it run `pip install -U langchain-ch
roma` and import as `from langchain_chroma import Chroma`.
  warn_deprecated(
Response: The provided text states "Addams Family Reunion The Addams Family goes on a search for their relatives."

Therefore, based on this information, we can say that the Addams Family are a group of characters (likely fictional
) who are searching for their relatives.


Let me know if you have any other questions!

Sources: ['None:None:2', 'None:None:0', 'None:None:3', 'None:None:1', 'None:None:4']
```

Test for Hallucinations

Test 2: Movie that is **NOT** listed should **NOT** return any text related to that movie

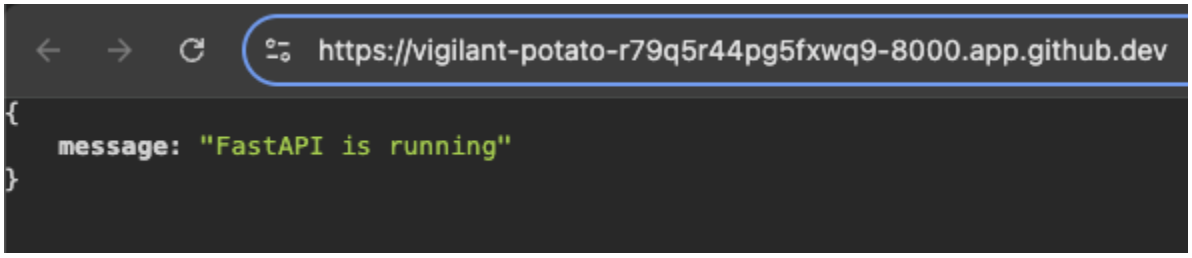
- In the same terminal, type the following:
  - **python query_data.py "What is <any_movie_title_listed> movie?"**
  - **Example:**
    - **python query_data.py "What is Alien movie?"**

```
@clarkngo → /workspaces/DIT637-TT09/modelserver-fastapi (main) $ python query_data.py "What is Alien movie?"
/usr/local/python/3.12.1/lib/python3.12/site-packages/langchain_core/_api/deprecation.py:141: LangChainDeprecationW
arning: The class `Chroma` was deprecated in LangChain 0.2.9 and will be removed in 0.4. An updated version of the
class exists in the langchain-chroma package and should be used instead. To use it run `pip install -U langchain-ch
roma` and import as `from langchain_chroma import Chroma`.
  warn_deprecated(
Response: The provided text does not contain information about the "Alien" movie.  It appears to be a collection of
 different titles related to cinema and familial relationships.

Sources: ['None:None:0', 'None:None:3', 'None:None:2', 'None:None:1', 'None:None:4']
```

Take a screenshot of your 'MOVIE NO INFO' as 'firstname_lastname_movie_no_info.png'.

- In the same terminal:
  - **uvicorn main:app --reload**
    The 'uvicorn main:app –reload' is used to start a FastAPI application with Uvicorn, an ASGI (Asynchronous Server Gateway Interface) server.
- Copy the Forwarded Address (URL)
- Do a quick test if its running

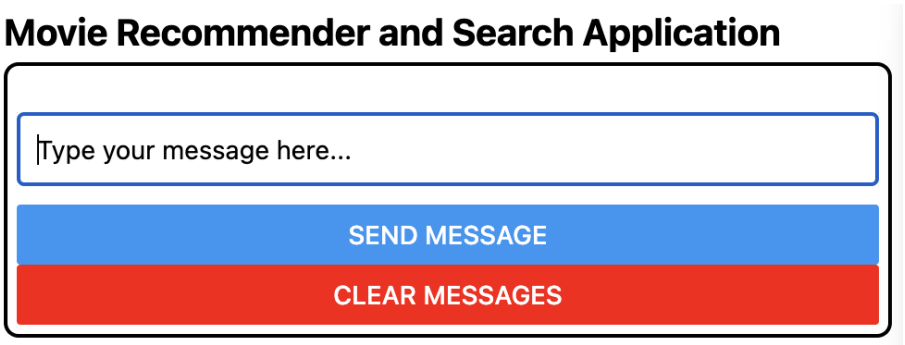


### *2.6) Running the Frontend React Native Mobile/Web*

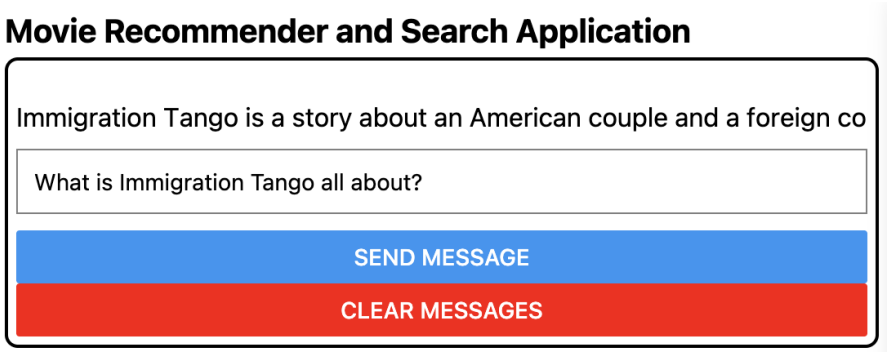
- Create a **.env** file inside 'frontend-reactnative' folder (use the 'example.env' file as a reference)
  - Update the **API_URL= with ExpressJS Forwarded Address**
  - Please be mindful to NOT include a "/" **at the end**

```
frontend-reactnative > .env
1   # ExpressJS Forwarded Address
2   API_URL=https://automatic-space-lamp-r6g47v59gq925r5x-3000.app.github.dev
3
```

- Open a new terminal:
  - **cd frontend-reactnative**
  - **npm install**
  - **npx expo start --web** (or if mobile: npx expo start --tunnel)
- Open the Forwarded Address.

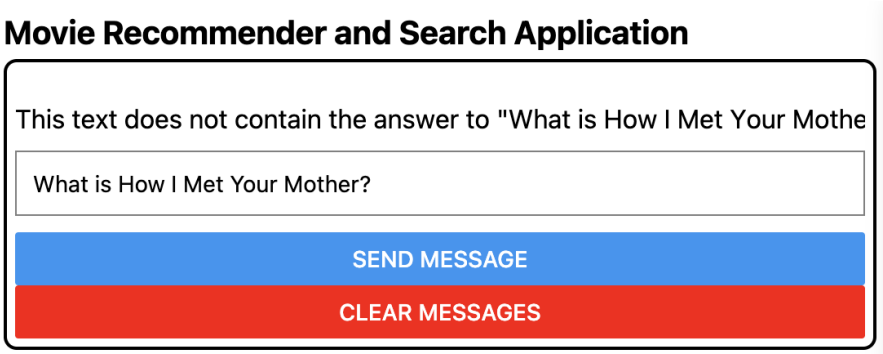



- From your list of movies, ask about a different movie
- Try to chat and send messages. For example:
  - "What Immigration Tango all about?"

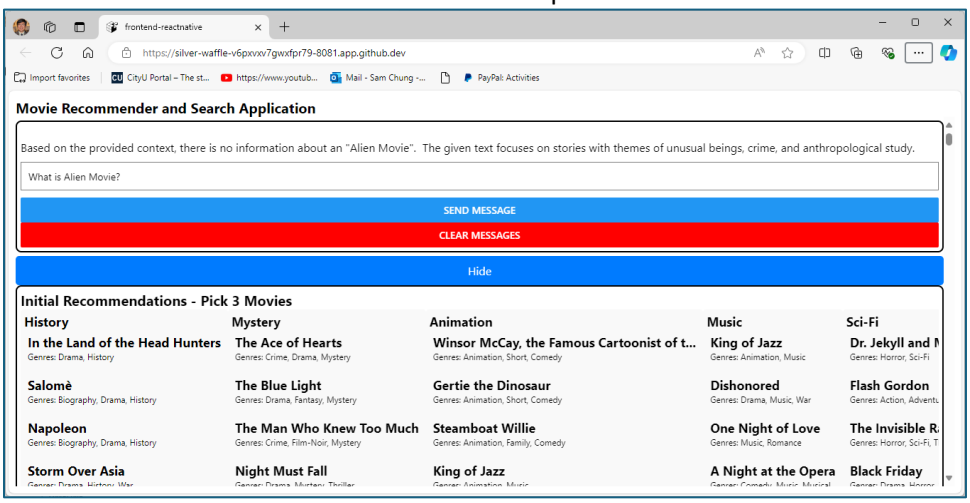


Take a screenshot of your ‘MOVIE HAS INFO’ as ‘firstname_lastname_has_info.png’.

- Clear the message and ask anything. It should not attempt to provide an answer.

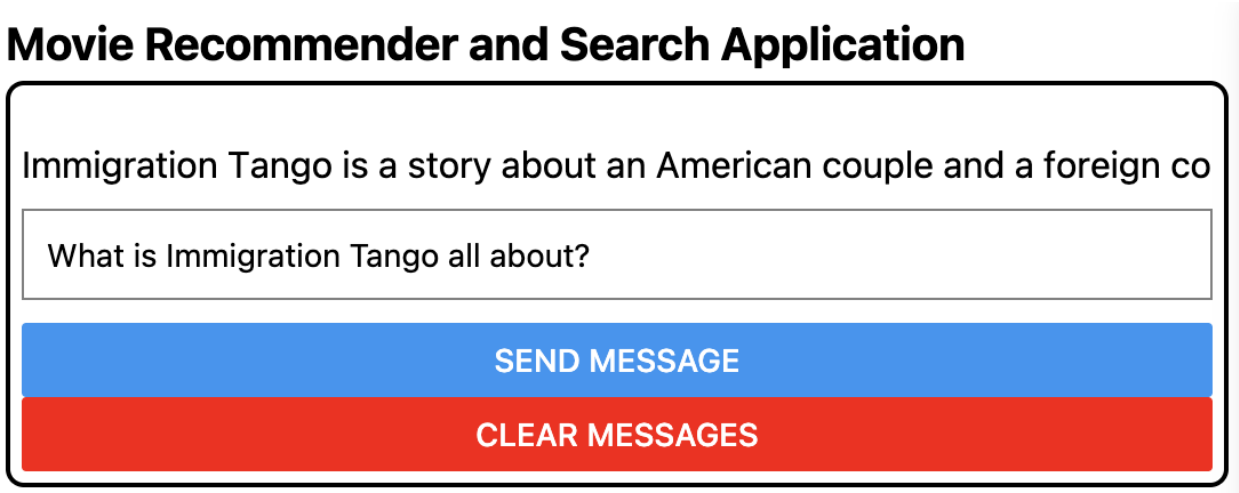



Another Example



### 3) Screenshot Summary

Chat: test the chat with asking for movie information. For example: "What is Immigration Tango all about?".



## 4) Code Breakdown – Python Scripts
### Populate Database (populate_database.py)

*Import Statement*

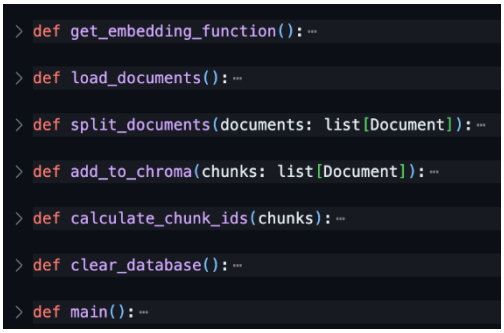
- argparse: Handles command-line arguments, allowing for database clearing.
- os: Provides operating system-related functions.
- shutil: Offers utility functions for file and directory operations.
- OllamaEmbeddings: Embeds text using the Ollama model.
- RecursiveCharacterTextSplitter: Splits text into chunks based on character count.
- Document: Represents a text document with metadata.
- Chroma: A vector database for storing and retrieving embeddings.
- requests: Makes HTTP requests to fetch data.
- pymongo: Interacts with MongoDB databases (if needed).

```
import argparse
import os
import shutil
from langchain_community.embeddings.ollama import OllamaEmbeddings
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain.schema.document import Document
from langchain_community.vectorstores import Chroma

import requests
import pymongo
```

*Functions Defined*

- get_embedding_function(): Returns the Ollama embeddings function.
- load_documents(): Fetches movie data from an API, converts it into a list of Document objects, and returns the list.
- split_documents(): Splits the given documents into chunks using the RecursiveCharacterTextSplitter.
- add_to_chroma(): Loads the existing Chroma database, calculates chunk IDs, adds new chunks to the database, and persists changes.
- calculate_chunk_ids(): Assigns unique IDs to each chunk based on its source, page number, and index within the page.
- clear_database(): Removes the Chroma database directory.
- main(): The main entry point, handling command-line arguments and calling other functions.




### *Embeddings Function*

The get_embedding_function() function provides a convenient way to obtain an embedding function using the Ollama embeddings model, which can then be used to embed text and perform tasks related to semantic similarity.

- OllamaEmbeddings is a class or function from the LangChain library that provides access to the Ollama embeddings model.
- model="gemma2:2b" specifies the particular Ollama model to be used. In this case, it's the "gemma2:2b" model.
- The embeddings variable is assigned an instance of the OllamaEmbeddings class, representing the embeddings model.

```
def get_embedding_function():
    embeddings = OllamaEmbeddings(model="gemma2:2b")
    return embeddings
```

### *Loading documents*

This function fetches movie data from an API, extracts the title and plot for each movie, creates Document objects with this information, and returns a list of these Document objects. These Document objects can then be used for further processing, such as embedding or storing in a vector database.

- response = requests.get('http://127.0.0.1:3000/movies-sample'):
  - This line makes a GET request to the URL http://127.0.0.1:3000/movies-sample. This is an API endpoint that provides a list of movies.
  - The requests library is used to make HTTP requests.
  - The response from the API is stored in the response variable.
- movies = response.json():
  - This line assumes that the API response is in JSON format and converts it into a Python dictionary or list.
  - The converted data is stored in the ‘movies’ variable.
- documents = []:
  - This line initializes an empty list named documents to store the converted movie data as Document objects.
- for movie in movies:
  - This line starts a loop that iterates over each movie in the movies list.
- if 'title' in movie and 'plot' in movie:
  - This line checks if the current movie contains both the title and plot keys. If both keys are present, the code continues.
- print(movie['title']):

- This line prints the title of the current movie to the console.
- text = f"{movie['title']} {movie['plot']}":
  - This line creates a string text that combines the title and plot of the current movie. The f-string syntax is used to format the string.
- documents.append(Document(page_content=text, metadata={'movie_id': movie['_id']})):
  - This line creates a new Document object with the following properties:
    - page_content: The combined title and plot text.
    - metadata: A dictionary containing metadata about the document:
      - movie_id: The unique identifier of the movie.

```
def load_documents():
    response = requests.get('http://127.0.0.1:3000/movies-sample')
    movies = response.json()

    documents = []
    for movie in movies:
        if 'title' in movie and 'plot' in movie:
            print(movie['title'])
            text = f"{movie['title']} {movie['plot']}"
            documents.append(Document(page_content=text, metadata={'movie_id': movie['_id']}))

    return documents
```

*Split Documents*

This function takes a list of Document objects as input and splits them into smaller chunks using the RecursiveCharacterTextSplitter. The resulting chunks can then be used for further processing, such as embedding or storing in a vector database.

- text_splitter = RecursiveCharacterTextSplitter(chunk_size=800, chunk_overlap=80, length_function=len, is_separator_regex=False):
  - This line creates an instance of the RecursiveCharacterTextSplitter class from the LangChain library.
  - The RecursiveCharacterTextSplitter is designed to split text into chunks based on character count.
- The following parameters are passed to the constructor:
  - chunk_size=800: Specifies that each chunk should be approximately 800 characters long.
  - chunk_overlap=80: Indicates that there should be an overlap of 80 characters between adjacent chunks to maintain context.
  - length_function=len: Uses the built-in len function to determine the length of text.
  - is_separator_regex=False: Indicates that the text splitter should not use regular expressions to determine chunk boundaries.
- return text_splitter.split_documents(documents):
  - This line calls the split_documents method of the text_splitter object, passing the documents list as an argument.
  - The split_documents method splits the given documents into smaller chunks based on the specified parameters and returns the resulting list of chunks.

```python
def split_documents(documents: list[Document]):
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=800,
        chunk_overlap=80,
        length_function=len,
        is_separator_regex=False,
    )
    return text_splitter.split_documents(documents)
```

*Calculate Chunk IDs*

This function loads an existing Chroma database, calculates IDs for the given chunks, identifies any new chunks, adds them to the database, and persists the changes.

- last_page_id = None:
  - This line initializes a variable named last_page_id to None, which will be used to keep track of the ID of the last processed page.
- current_chunk_index = 0:
  - This line initializes a variable named current_chunk_index to 0, which will be used to keep track of the index of the current chunk within a page.
- for chunk in chunks::
  - This line starts a loop that iterates over each chunk in the chunks list.
- source = chunk.metadata.get("source"):
  - This line extracts the source value from the metadata dictionary of the current chunk. The get method is used to avoid raising an error if the source key is not present.
- page = chunk.metadata.get("page"):
  - This line extracts the page value from the metadata dictionary of the current chunk.
- current_page_id = f"{source}:{page}":
  - This line creates a string current_page_id that combines the source and page values, separated by a colon. This string will be used to identify the current page.
- if current_page_id == last_page_id::
  - This line checks if the current_page_id is the same as the last_page_id. If they are the same, it means that the current chunk is part of the same page as the previous chunk.
- current_chunk_index += 1:
  - If the current_page_id is the same as the last_page_id, this line increments the current_chunk_index to indicate that this is the next chunk within the same page.
- else::
  - If the current_page_id is different from the last_page_id, it means that the current chunk is part of a new page.
- current_chunk_index = 0:
  - If the current_page_id is different from the last_page_id, this line resets the current_chunk_index to 0, indicating that this is the first chunk of the new page.
- chunk_id = f"{current_page_id}:{current_chunk_index}":
  - This line creates a string chunk_id that combines the current_page_id and the current_chunk_index, separated by a colon. This string will be used as the unique ID for the current chunk.
- last_page_id = current_page_id:
  - This line updates the last_page_id to the current_page_id for the next iteration.
- chunk.metadata["id"] = chunk_id:

- This line adds the chunk_id to the metadata dictionary of the current chunk, making it accessible for later use.

- return chunks:
  - This line returns the chunks list, with each chunk now containing a unique id in its metadata.

```python
def calculate_chunk_ids(chunks):

    last_page_id = None
    current_chunk_index = 0

    for chunk in chunks:
        source = chunk.metadata.get("source")
        page = chunk.metadata.get("page")
        current_page_id = f"{source}:{page}"

        # If the page ID is the same as the last one, increment the index.
        if current_page_id == last_page_id:
            current_chunk_index += 1
        else:
            current_chunk_index = 0

        # Calculate the chunk ID.
        chunk_id = f"{current_page_id}:{current_chunk_index}"
        last_page_id = current_page_id

        # Add it to the page meta-data.
        chunk.metadata["id"] = chunk_id

    return chunks
```

### *Add Documents to ChromaDB*

This function loads an existing Chroma database, calculates IDs for the given chunks, identifies any new chunks, adds them to the database, and persists the changes.

- db = Chroma(persist_directory=CHROMA_PATH, embedding_function=get_embedding_function()):
  - This line creates an instance of the Chroma class from the LangChain library.
  - The Chroma class is used to manage a vector database for storing and retrieving embeddings.
  - The persist_directory parameter specifies the directory where the Chroma database should be stored.
  - The embedding_function parameter is used to provide the embedding function that will be used to create embeddings for the documents.
- chunks_with_ids = calculate_chunk_ids(chunks):
  - This line calls the calculate_chunk_ids function to assign unique IDs to each chunk in the chunks list. The resulting list of chunks with IDs is stored in the chunks_with_ids variable.
- existing_items = db.get(include=[]):
  - This line retrieves all existing items from the Chroma database. The include=[] argument ensures that all fields, including the item IDs, are included in the results.
  - The retrieved items are stored in the existing_items variable.
- existing_ids = set(existing_items["ids"]):
  - This line extracts the IDs of all existing items from the existing_items dictionary and creates a set of these IDs. The set is stored in the existing_ids variable.
- new_chunks = []:

    - This line initializes an empty list named new_chunks to store the chunks that need to be added to the database.
- for chunk in chunks_with_ids::
    - This line starts a loop that iterates over each chunk in the chunks_with_ids list.
- if chunk.metadata["id"] not in existing_ids::
    - This line checks if the ID of the current chunk is already in the existing_ids set. If it's not, the chunk is a new one and should be added to the database.
- new_chunks.append(chunk):
    - If the chunk is new, this line appends it to the new_chunks list.
- if len(new_chunks)::
    - This line checks if there are any new chunks to be added.
- new_chunk_ids = [chunk.metadata["id"] for chunk in new_chunks]:
    - This line creates a list of IDs for the new chunks.
- db.add_documents(new_chunks, ids=new_chunk_ids):
    - This line adds the new chunks to the Chroma database. The ids argument is used to provide the corresponding IDs for the chunks.
- db.persist():
    - This line persists the changes made to the Chroma database to ensure that they are saved to disk.

```python
def add_to_chroma(chunks: list[Document]):
    # Load the existing database.
    db = Chroma(
        persist_directory=CHROMA_PATH, embedding_function=get_embedding_function()
    )

    # Calculate Page IDs.
    chunks_with_ids = calculate_chunk_ids(chunks)

    # Add or Update the documents.
    existing_items = db.get(include=[])  # IDs are always included by default
    existing_ids = set(existing_items["ids"])
    print(f"Number of existing documents in DB: {len(existing_ids)}")

    # Only add documents that don't exist in the DB.
    new_chunks = []
    for chunk in chunks_with_ids:
        if chunk.metadata["id"] not in existing_ids:
            new_chunks.append(chunk)

    if len(new_chunks):
        print(f"👉 Adding new documents: {len(new_chunks)}")
        new_chunk_ids = [chunk.metadata["id"] for chunk in new_chunks]
        db.add_documents(new_chunks, ids=new_chunk_ids)
        db.persist()
        print(f"✅ Finished adding new documents: {len(new_chunks)}")
    else:
        print("✅ No new documents to add")
```

### *Clear Database*

This function is designed to remove the Chroma database directory, effectively clearing its contents. It's often used to reset the database when necessary.

- def clear_database():: This line defines a Python function named clear_database.

- if os.path.exists(CHROMA_PATH)::
  - os.path.exists(CHROMA_PATH) checks if the directory specified by CHROMA_PATH exists.
- shutil.rmtree(CHROMA_PATH):
  - If the directory exists, this line uses the shutil.rmtree function to remove the directory and all of its contents recursively. This means that it will delete the directory itself, as well as any subdirectories and files within it.

```
def clear_database():
    if os.path.exists(CHROMA_PATH):
        shutil.rmtree(CHROMA_PATH)
```

*Main Function*

This function is named main and serves as the entry point for the Python script. It's the first function that is executed when the script is run.

- parser = argparse.ArgumentParser():
  - This line creates an instance of the ArgumentParser class from the argparse module. This class is used to parse command-line arguments.
- parser.add_argument("--reset", action="store_true", help="Reset the database."):
  - This line adds a command-line argument named --reset to the parser. The action="store_true" argument indicates that this argument is a flag, and its value will be True if it's present in the command-line arguments. The help argument provides a brief description of the argument.
- args = parser.parse_args():
  - This line parses the command-line arguments and stores the parsed arguments in the args variable.
- if args.reset::
  - This line checks if the --reset argument was present in the command-line arguments. If it was, the code within the if block will be executed.
- print("✨ Clearing Database"):
  - If the --reset argument was present, this line prints a message indicating that the database will be cleared.
- clear_database():
  - If the --reset argument was present, this line calls the clear_database function to remove the Chroma database directory.
- documents = load_documents():
  - This line calls the load_documents function to load documents from an API or other source.
- chunks = split_documents(documents):
  - This line calls the split_documents function to split the loaded documents into smaller chunks.
- add_to_chroma(chunks):
  - This line calls the add_to_chroma function to add the chunks to the Chroma database.

```python
def main():
    # Check if the database should be cleared (using the --clear flag).
    parser = argparse.ArgumentParser()
    parser.add_argument("--reset", action="store_true", help="Reset the database.")
    args = parser.parse_args()
    if args.reset:
        print("✨ Clearing Database")
        clear_database()

    # Create (or update) the data store.
    documents = load_documents()
    chunks = split_documents(documents)
    add_to_chroma(chunks)
```

### Query Data (query_data.py)

*Query with RAG*

This function implements a Retrieval Augmented Generation (RAG) approach to process a query. It retrieves relevant documents from a Chroma database, creates a prompt using the query and the retrieved documents, and then generates a response using a language model. The function also provides information about the sources used to generate the response.

- embedding_function = get_embedding_function():
  - This line calls the get_embedding_function function to obtain an embedding function. This function is likely defined elsewhere and returns a function that can be used to create embeddings for text.
- db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedding_function):
  - This line creates an instance of the Chroma class from the LangChain library, which is used to manage a vector database for storing and retrieving embeddings.
  - The persist_directory parameter specifies the directory where the Chroma database should be stored.
  - The embedding_function parameter is used to provide the embedding function that will be used to create embeddings for the documents.
- results = db.similarity_search_with_score(query_text, k=5):
  - This line performs a similarity search in the Chroma database using the query_text. The k=5 argument specifies that the search should return the top 5 most similar documents.
  - The results of the search are stored in the results variable, which is a list of tuples containing the document and its similarity score.
- context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in results]):
  - This line concatenates the page content of the top 5 documents into a single string, separated by "\n\n---\n\n". This string will be used as the context for the subsequent generation task.
- prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE):
  - This line creates a ChatPromptTemplate object from a template named PROMPT_TEMPLATE. This template is likely defined elsewhere and contains the structure of the prompt that will be used to generate a response.
- prompt = prompt_template.format(context=context_text, question=query_text):
  - This line formats the prompt_template using the context_text and query_text as placeholders. The resulting formatted prompt will be used to query the language model.
- model = Ollama(model="gemma2:2b"):
  - This line creates an instance of the Ollama class from the LangChain library, which is used to interact with the Ollama language model.

    - The model="gemma2:2b" argument specifies the particular Ollama model to be used.
- response_text = model.invoke(prompt):
    - This line invokes the Ollama language model with the formatted prompt and stores the generated response in the response_text variable.
- sources = [doc.metadata.get("id", None) for doc, _score in results]:
    - This line extracts the IDs of the top 5 documents from the results list and stores them in the sources list.
- formatted_response = f"Response: {response_text}\nSources: {sources}":
    - This line creates a formatted string that includes the generated response and the sources used to generate it.
- print(formatted_response):
    - This line prints the formatted response to the console.
- return response_text:
    - This line returns the generated response text.

```python
def query_rag(query_text: str):
    # Prepare the DB.
    embedding_function = get_embedding_function()
    db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedding_function)

    # Search the DB.
    results = db.similarity_search_with_score(query_text, k=5)

    context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in results])
    prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
    prompt = prompt_template.format(context=context_text, question=query_text)

    model = Ollama(model="gemma2:2b")
    response_text = model.invoke(prompt)

    sources = [doc.metadata.get("id", None) for doc, _score in results]
    formatted_response = f"Response: {response_text}\nSources: {sources}"
    print(formatted_response)
    return response_text
```

*Main Function*

The main function creates a command-line interface using argparse to allow the user to provide a query text. It then calls the query_rag function to process the query and generate a response.

- parser = argparse.ArgumentParser():
    - This line creates an instance of the ArgumentParser class from the argparse module. This class is used to parse command-line arguments.
- parser.add_argument("query_text", type=str, help="The query text."):
    - This line adds a positional argument named query_text to the parser. The type=str argument specifies that the value of this argument should be a string. The help argument provides a brief description of the argument.
- args = parser.parse_args():

    - This line parses the command-line arguments and stores the parsed arguments in the args variable.
- query_text = args.query_text:
    - This line extracts the value of the query_text argument from the args object and assigns it to the query_text variable.
- query_rag(query_text):
    - This line calls the query_rag function, passing the query_text as an argument. The query_rag function is presumably responsible for processing the query and generating a response.

```
def main():
    # Create CLI.
    parser = argparse.ArgumentParser()
    parser.add_argument("query_text", type=str, help="The query text.")
    args = parser.parse_args()
    query_text = args.query_text
    query_rag(query_text)
```

## 5) Code Breakdown – Model Server FastAPI

*Query RAG*

This function is named query_rag and is designed to process a given query text using a Retrieval Augmented Generation (RAG) approach.

- embedding_function = get_embedding_function():
  - This line calls the get_embedding_function function to obtain an embedding function. This function is likely defined elsewhere and returns a function that can be used to create embeddings for text.
- db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedding_function):
  - This line creates an instance of the Chroma class from the LangChain library, which is used to manage a vector database for storing and retrieving embeddings.
  - The persist_directory parameter specifies the directory where the Chroma database should be stored.
  - The embedding_function parameter is used to provide the embedding function that will be used to create embeddings for the documents.
- results = db.similarity_search_with_score(query_text, k=5):
  - This line performs a similarity search in the Chroma database using the query_text. The k=5 argument specifies that the search should return the top 5 most similar documents.
  - The results of the search are stored in the results variable, which is a list of tuples containing the document and its similarity score.
- context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in results]):
  - This line concatenates the page content of the top 5 documents into a single string, separated by "\n\n---\n\n". This string will be used as the context for the subsequent generation task.
- prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE):
  - This line creates a ChatPromptTemplate object from a template named PROMPT_TEMPLATE. This template is likely defined elsewhere and contains the structure of the prompt that will be used to generate a response.
- prompt = prompt_template.format(context=context_text, question=query_text):
  - This line formats the prompt_template using the context_text and query_text as placeholders. The resulting formatted prompt will be used to query the language model.
- model = Ollama(model="gemma2:2b"):
  - This line creates an instance of the Ollama class from the LangChain library, which is used to interact with the Ollama language model.
  - The model="gemma2:2b" argument specifies the particular Ollama model to be used.
- response_text = model.invoke(prompt):
  - This line invokes the Ollama language model with the formatted prompt and stores the generated response in the response_text variable.
- return response_text:
  - This line returns the generated response text.

```python
def query_rag(query_text: str):
    # Prepare the DB.
    embedding_function = get_embedding_function()
    db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedding_function)

    # Search the DB.
    results = db.similarity_search_with_score(query_text, k=5)

    context_text = "\n\n---\n\n".join([doc.page_content for doc, _score in results])
    prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
    prompt = prompt_template.format(context=context_text, question=query_text)

    model = Ollama(model="gemma2:2b")
    response_text = model.invoke(prompt)

    return response_text
```

### *Route /chat*

This code defines an endpoint that handles chat requests, retrieves a response using the query_rag function, and returns the response as JSON. In case of any errors, it raises a 500 status code with error details for debugging purposes.

- @app.post("/chat"):
  - This line is a decorator that registers this function as the handler for POST requests to the /chat URL.
- def chat_endpoint(request: QueryRequest):
  - This line defines the chat_endpoint function that takes a single argument, request, of type QueryRequest (likely a custom class defined elsewhere).
- try:
  - This block marks the beginning of a try block to handle potential exceptions.
- response = query_rag(request.query_text):
  - This line calls the query_rag function, passing the query_text attribute of the request object as an argument. The query_rag function is assumed to be defined elsewhere and is responsible for processing the query text and generating a response.
- return {"response": response}:
  - If the query_rag function executes successfully, this line returns a dictionary with a single key-value pair: response: response. This dictionary will be serialized as JSON and sent back to the client.
- except Exception as e:
  - This block defines an exception handler using except Exception as e:. This will catch any exception that might occur within the try block.
- raise HTTPException(status_code=500, detail=str(e)):
  - If an exception is caught, this line raises a new HTTPException with a status code of 500 (Internal Server Error) and a detailed message containing the string representation of the exception (str(e)). This ensures that the client receives an appropriate error response.

```python
@app.post("/chat")
def chat_endpoint(request: QueryRequest):
    try:
        response = query_rag(request.query_text)
        return {"response": response}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))
```

### 6) Code Breakdown – Proxy & API Server ExpressJS

This code snippet defines a route handler for a POST request to the /chat endpoint using an ExpressJS. It simulates streaming a response from a FastAPI server located at http://localhost:8000. It demonstrates how to create a simulated streaming response for a chat request. It first validates the input, prepares for streaming headers, creates a PassThrough stream, fetches the actual response from a FastAPI server, and then simulates sending the response in chunks. It handles potential errors by sending an error response.

- **app.post('/chat', async (req, res) => { ... }):**
  - This line defines an asynchronous route handler for the POST request to the /chat endpoint. The async keyword indicates that the function can use asynchronous operations (like waiting for network responses).
- **const { query_text } = req.body;:**
  - This line extracts the query_text property from the request body (req.body) and assigns it to a constant named query_text.
- **if (!query_text) { ... }:**
  - This block checks if the query_text is missing from the request body.
  - If it's missing, it sends a response with a status code of 400 (Bad Request) and an error message indicating that query_text is required.
- **res.setHeader('Content-Type', 'text/plain; charset=utf-8');:**
  - This line sets the Content-Type header of the response to text/plain; charset=utf-8, indicating that the response is plain text encoded in UTF-8.
- **res.setHeader('Transfer-Encoding', 'chunked');:**
  - This line sets the Transfer-Encoding header to chunked, indicating that the response will be sent in multiple chunks. This allows the client to receive and process the response as it arrives, simulating a streaming experience.
- **const stream = new PassThrough();:**
  - This line creates a new PassThrough stream object. This type of stream allows data to be written to it and will be passed through to the destination (in this case, the response object res).
- **stream.pipe(res);:**
  - This line pipes the stream to the res object. This means any data written to the stream will be sent as part of the response.
- **try { ... } catch (error) { ... }:**
  - This block wraps the main logic in a try...catch block to handle any potential errors.
- **const fastApiResponse = await axios.post('http://localhost:8000/chat', { ... });:**
  - This line uses a library like axios to make an asynchronous POST request to the FastAPI server at http://localhost:8000/chat.
  - It sends a payload containing the query_text extracted from the initial request.
  - The await keyword pauses the execution of the function until the API response is received.
- **const responseText = fastApiResponse.data.response;:**
  - This line extracts the response property from the response data (fastApiResponse.data) of the FastAPI request. This assumes the FastAPI response is a JSON object containing a response field.
- **for (let i = 0; i < responseText.length; i++) { ... }:**
  - This loop iterates over each character in the responseText.
  - Inside the loop:
    - The character is written to the stream using stream.write(responseText[i]).

    - A short delay of 50 milliseconds is simulated using await new Promise(resolve => setTimeout(resolve, 50)). This is just for demonstration purposes and wouldn't be used in a real streaming scenario.
- **stream.end();:**
    - After the loop is finished, this line calls stream.end(), indicating that there's no more data to be written.
- **catch (error) { ... }:**
    - If any error occurs during the process (e.g., network issues or errors in the FastAPI response), this block will handle it.
    - It closes the stream (stream.end()) and sends an error response with a status code of 500 (Internal Server Error) and a generic error message.

```
app.post('/chat', async (req, res) => {
  const { query_text } = req.body;

  // Validate the input
  if (!query_text) {
    return res.status(400).send({ error: "query_text is required" });
  }

  // Set headers for streaming
  res.setHeader('Content-Type', 'text/plain; charset=utf-8');
  res.setHeader('Transfer-Encoding', 'chunked');

  // Create a PassThrough stream
  const stream = new PassThrough();
  stream.pipe(res);

  try {
    // Call the FastAPI /chat endpoint
    const fastApiResponse = await axios.post('http://localhost:8000/chat', {
      query_text: query_text
    });

    const responseText = fastApiResponse.data.response;

    // Simulate streaming by sending chunks of text
    for (let i = 0; i < responseText.length; i++) {
      stream.write(responseText[i]);
      await new Promise(resolve => setTimeout(resolve, 50)); // Simulate delay
    }

    stream.end();
  } catch (error) {
    stream.end();
    res.status(500).send({ error: "Error processing request" });
  }
});
```

## 7) Code Breakdown – Frontend React Native Chat

*Chat Function*

This function handles sending a chat request, receiving the response in chunks, and updating the application state with the received messages. It ensures proper decoding of chunks and updates the UI incrementally as the response arrives.

- Setting Loading State:
  - setIsLoading(true); sets a loading state to true, indicating that the application is waiting for a response from the chat server. This is likely used to display a loading indicator to the user.
- Fetching Chat Response:
  - try...catch block: This block handles potential errors that might occur during the chat interaction.
  - const response = await fetch(...):
    - An asynchronous fetch request is made to the /chat endpoint at the specified API_URL.
    - The request method is set to POST as it's sending data to the server.
    - The Content-Type header is set to application/json to indicate that the request body is JSON data.
  - The request body is a JSON object containing the query_text property set to the user's input (userInput).
  - if (response.ok) { ... }:
    - This block executes only if the fetch request is successful (status code in the 200 range).
- Handling Streaming Response:
  - const reader = response.body.getReader();:
    - A ReadableStream object is retrieved from the response body. This allows reading the response data in chunks.
  - const decoder = new TextDecoder('utf-8');:
    - A TextDecoder object is created to decode the response data from bytes to UTF-8 text.
  - let accumulatedResponse = '';:
    - An empty string variable accumulatedResponse is initialized to store the complete response as it's received in chunks.
  - while (true) { ... }:
    - An infinite loop continues until the response is fully received (controlled by the done flag).
    - Inside the loop:
      - const { done, value } = await reader.read();
        - The read() method of the reader is called asynchronously. It returns an object with two properties:
          - done: A boolean indicating if there's more data to be read.
          - value: A Uint8Array containing the next chunk of data.
      - if (done) break;: If done is true, it means all data has been read, and the loop exits.
      - const decodedChunk = decoder.decode(value, { stream: true });
        - The value (chunk) is decoded using the text decoder. The stream: true option ensures proper handling of incomplete UTF-8 characters that might span across chunks.
      - accumulatedResponse += decodedChunk;
        - The decoded chunk is appended to the accumulatedResponse string.

        - setMessages((prevMessages) => [...prevMessages, decodedChunk]);
    - This line likely updates the application state with the received chunk. It's assumed there's a state variable called messages that holds an array of chat messages. The update function adds the new chunk (decodedChunk) to the existing messages array.
- Error Handling:
    - catch (error) { ... }:
        - If an error occurs during the fetch or processing, this block handles it.
        - console.error('Error fetching chat:', error); logs the error message to the console.
- Setting Loading State (Again):
    - setIsLoading(false); sets the loading state to false, indicating that the chat response has been received and processed.

```
const handleChat = async () => {
    setIsLoading(true);

    try {
        const response = await fetch(`${API_URL}/chat`, {
            method: 'POST',
            headers: {
                'Content-Type': 'application/json',
            },
            body: JSON.stringify({
                query_text: userInput,  // Sending user input as query_text
            }),
        });
```

```
if (response.ok) {
    const reader = response.body.getReader();
    const decoder = new TextDecoder('utf-8');

    let accumulatedResponse = ''; // Accumulate the entire response

    while (true) {
        const { done, value } = await reader.read();
        if (done) break;

        // Decode the stream chunk
        const decodedChunk = decoder.decode(value, { stream: true });

        // Accumulate the response
        accumulatedResponse += decodedChunk;

        // Update the messages with the new chunk
        setMessages((prevMessages) => [...prevMessages, decodedChunk]);
    }
}
    } catch (error) {
        console.error('Error fetching chat:', error);
    }
    setIsLoading(false);
};
```

**What is Streaming and Chunking?**
Streaming delivers data in small, manageable pieces (chunks) as it's produced, rather than waiting for the entire data set.

**Streaming**, a process of continuous data transmission from a source to a destination, offers real-time benefits. It allows data to be processed and consumed as it arrives, making it particularly engaging for applications like video streaming, live broadcasts, and online gaming.

**Chunking** is a data transfer method in which the data stream is divided into smaller, manageable pieces called “chunks.” Each chunk is preceded by its size in bytes, and the transmission ends when a zero-length chunk is received. This method is particularly useful when the content length is not known beforehand.

This allows immediate processing and display of information, reducing latency and memory usage, especially useful for large files or real-time data like video, audio, or continuous text responses.

The code creates a chat interface using React Native components.

- It includes a View container with a FlatList to display chat messages. The list is rendered horizontally, with a separator between items.
- A TextInput allows users to type their messages, and a Button sends the message when pressed.
- A Button to clear the message when pressed.
- The FlatList uses index as a key for simplicity, and the Button is disabled while loading.

```
{/* Chat  */}
<View style={styles.searchContainer}>
    <FlatList
        data={messages}
        renderItem={renderMessage}
        keyExtractor={(item, index) => index.toString()} // Use index as key for simplicity
        horizontal // Enable horizontal scrolling
        style={styles.messageList}
    />
    <TextInput
        style={styles.input}
        placeholder="Type your message here..."
        value={userInput}
        onChangeText={text => setUserInput(text)}
    />
    <Button title="Send Message" onPress={handleChat} disabled={isLoading} />
    <Button title="Clear Messages" onPress={clearMessages} color="red" />
</View>
```

### 8) Pushing your work to GitHub

1. Go to Source Control on your GitHub codespaces and observe the pending changes.

2. Type the Message for your changes in the Message box on the top. For example," **Submission for TT09 – Your Name**"

3. Click on the dropdown beside the commit button and select **Commit & Push** to update the changes to your repository main branch.

4. Select **Yes** when prompted.