Suhas Ramesha

Problem Statement-1 :

1. API Data Retrieval and Storage

For this problem, the task is to get book data from an API and store it in a local database. I am assuming the API gives the data in JSON format and contains simple fields like book name, author, and year. First, I would call the API using Python and check if the response is coming properly. After that, I would convert the response into JSON so that I can read it easily in Python.

Example code for reading API data:

```
import requests
res = requests.get(api_url)
data = res.json()
```

Once I have the data, I would create a small SQLite database file and a table for storing the book details. Then, I would loop through the API data and insert each book into the database one by one. After inserting, I would fetch the data again from the database just to confirm that everything is stored properly.

Example code for inserting into database:

```
for i in data:
cursor.execute("INSERT INTO books VALUES (?, ?, ?)",
(i["title"], i["author"], i["year"]))
```

2. Data Processing and Visualization

In this task, the objective is to work with student test score data fetched from an API and visualize it. I assume the API provides student names along with their respective scores in JSON format. After fetching the data, I would extract the scores and calculate the average using basic Python logic.

Example for calculating the average:

```
average_score = sum(scores) / len(scores)
```

To make the data easier to understand, I would visualize it using a bar chart where each bar represents a student's score. I would also add a reference line for the average score to give a better comparison across students. Visualization is useful here because it provides quick insights into performance differences.

Example for plotting:

```
plt.bar(names, scores)
plt.axhline(y=average_score, linestyle="--")
```

3. CSV Data Import into SQLite Database

In this problem, the task is to take user details from a CSV file and store them in a database. I am assuming the CSV file has columns like name and email. I would read the CSV file using pandas because it is easier to work with. After reading the file, I would loop through each row and insert the values into the database.

Example code part for reading CSV:

```
import pandas as pd
df = pd.read_csv("users.csv")
```

First, I would create a table in SQLite. Then, using a loop, I would add each user record from the CSV file into the table. After that, I would check the table to make sure the data is correctly inserted.

Example code part for inserting data:

```
for i in range(len(df)):
cursor.execute("INSERT INTO users VALUES (?, ?)",
(df.iloc[i]["name"], df.iloc[i]["email"]))
```

4. Complex Python code :

About the project :-
This is a chatbot project where we can upload a PDF and then ask questions from it. The bot reads the PDF and gives answers based on that file. It is using RAG concept, means it first finds related content and then answers.

Tech used :

Backend side:

- FastAPI is used to create APIs

- LangChain is used to connect LLM with documents

- LangGraph is used to manage the flow of chat and steps

- ChromaDB is used to store embeddings (vector database)

- Google Gemini model is used to generate answers

- HuggingFace embeddings are used to convert text into vectors

- PyPDF is used to read PDF files

- Uvicorn is used to run the backend server

Frontend side:

- React is used for UI

- Vite is used to run frontend faster

- Tailwind CSS is used for basic styling

- Axios is used to call backend APIs

- React Router is used for page navigation

How it works :

First, user uploads a PDF file. That PDF is saved and then split into small parts. After that, each part is converted into embeddings and stored in ChromaDB for that particular chat.

When the user asks a question, the system searches for similar chunks from the vector database. Then it creates a prompt using those chunks and previous chat messages. This prompt is sent to Gemini model and the answer is generated.

Chat history is stored in a JSON file so that previous messages are not lost.

Main features :

- Can upload PDF and ask questions

- Can handle multiple chats at same time

- Each chat has its own history

- Vector data is stored separately for each chat

- Chat flow is handled using LangGraph steps

Githhub Link :https://github.com/Suhas-Ramesha/Langgragh-RAG

5. Complex Backend Code :

About : This backend is built using FastAPI and is used to load already trained machine learning models for crypto price prediction. The API provides different endpoints to check server health, list available models, get model details for each coin, and make predictions. When a prediction request is sent, the backend takes historical data from the user, passes it to the ML model, and returns the predicted values. I also used Swagger UI to test the APIs easily. This backend was complex because it involved handling APIs, loading ML models, and managing different coins in a single system.

Link : https://github.com/Suhas-Ramesha/Crypto-Predict/tree/main/Milestone%203/backend

Problem Statement 2 :

1. Self-Rating on LLM, Deep Learning, AI, and ML

Based on my coursework, projects, and hands-on experience, I would rate myself as follows:

- **Machine Learning – A**
  I am comfortable with machine learning concepts and can build models independently. I have worked on data preprocessing, feature engineering, model training, evaluation, and basic optimization. I have implemented ML models as part of academic projects and backend systems.

- **Deep Learning – B**
  I have good understanding of deep learning fundamentals such as neural networks,

CNNs, and basic transformer concepts. I have implemented models using frameworks like PyTorch/TensorFlow and understand training, loss functions, and backpropagation. For complex architectures and tuning, I still explore and learn.
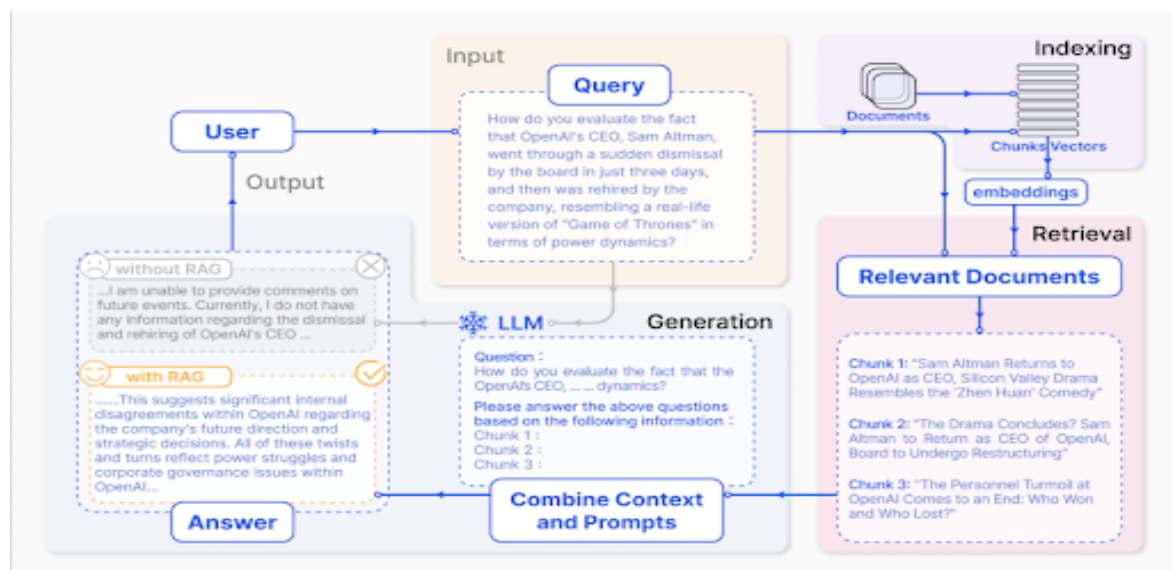
- **AI – B**
  I understand AI at a system level, including how ML, DL, and LLMs fit into real applications. I can design AI-based workflows and implement them, but advanced research-level AI still requires deeper study.

- **LLMs – A**
  I have worked on LLM-based chatbots and understand both fine-tuning and RAG-based approaches. I am familiar with how LLMs are integrated into backend systems, how prompts are constructed, and how external knowledge is injected using vector databases.

2. High-Level Architecture of an LLM-Based Chatbot

From my understanding and hands-on experience, an LLM-based chatbot usually follows **four main steps**: **Ingestion, Embedding, Retrieval, and Generation**. I have worked on these steps while building chatbot systems and during my experience at Vishwam AI.

1. Ingestion

In this step, the data is first collected. This can be PDFs, text files, or documents. I have worked with PDF and text-based data where the files are loaded and cleaned. Since large documents cannot be directly given to the model, the data is prepared for further processing.

During ingestion, the text is split into smaller parts so that it is easier to manage. This step is important because improper ingestion can affect the final answer quality.

2. Embedding (Chunking + Fine-Tuning Experience)

After ingestion, the text is split into **chunks**. I have worked with different chunking methods like fixed-size chunking, overlapping chunks, and basic semantic chunking. Overlapping chunks help in maintaining context between two consecutive parts.

Once chunking is done, each chunk is converted into embeddings using an embedding model. These embeddings represent the meaning of the text in numerical form and are stored in a vector database.

Along with embeddings, I have also worked on **fine-tuning an LLM using Unsloth**. In this process, I fine-tuned the model using instruction–response style data to improve how the chatbot responds. Fine-tuning helped in improving the response format and behavior, while embeddings were used for injecting external knowledge.

3. Retrieval

When a user asks a question, the query is converted into an embedding. This embedding is compared with stored embeddings in the vector database to find the most relevant chunks.

From my experience, this step is very important because the quality of retrieved chunks directly affects the final answer. I learned that selecting proper chunk size and retrieval count helps improve accuracy.

4. Generation

In the generation step, the retrieved chunks are added as context to a prompt. This prompt is then passed to the LLM. The LLM generates the final response based on the user question, retrieved content, and conversation history.

I learned that prompt structure and context size play a big role in answer quality. This step combines everything together to produce a meaningful response.

Experience at Vishwam AI

During my time at **Vishwam AI**, I worked on understanding and implementing this full 4-step pipeline in real chatbot systems. This experience helped me understand practical challenges like chunk overlap, embedding quality, retrieval accuracy, and response latency. It gave me a clear idea of how LLM-based chatbots work end-to-end in real applications.
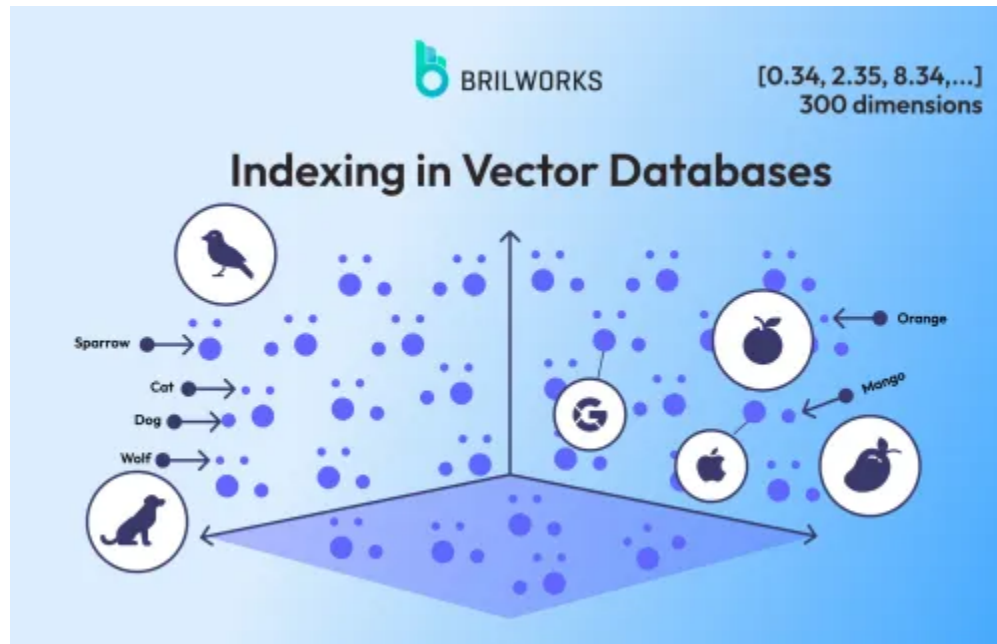
3. Vector Databases

A vector database is used to store embeddings and perform similarity search. Instead of matching exact words, it finds data that is similar in meaning. This is very important for LLM applications where semantic understanding is required.

I have studied and explored different vector database options:

- **Pinecone**
  Pinecone is a managed vector database that is easy to scale and suitable for production systems. It handles indexing, scaling, and performance automatically, which makes it useful for large-scale applications.

- **Supabase (pgvector)**
  Supabase uses PostgreSQL with the `pgvector` extension. This is useful when we want both relational data and vector search in the same database. It is good for projects that already rely on Postgres and need vector search as an added feature.

- **PostgreSQL**
  Using PostgreSQL directly with pgvector is a flexible option for self-hosted systems. It allows storing embeddings along with structured data and supports similarity search using SQL.

- **FAISS**
  FAISS is commonly used for local or research-level projects. It is fast and efficient but does not provide built-in persistence or scaling like managed databases.

Here's an Example of Data is stored in the Vector Database , where vector which are similar in value are placed near t each other and different vector values are placed far from each other :

Hypothetical Problem and Database Choice

Problem:
 Building a chatbot for internal company documents and security policies.

My choice:

- For a prototype or student project, I would use FAISS or Postgres + pgvector.

- For a production-level system, I would prefer Pinecone because of its scalability, performance, and managed infrastructure.

- If the system already uses Postgres for user and application data, Supabase with pgvector would be a practical choice.

The final choice depends on scale, cost, and system requirements.