**DESIGN:**

RAG: Retrival Agumented Generation

Architecture:

To represent the architecture of a Recommendation, Aggregation, and Generation (RAG) system. The system typically consists of components for data storage, a retrieval module, generation (e.g., language models), and an aggregation layer. RAG augments the capability of large language models with masses of data outside the base model, enabling model responses to generate more real, more individual, and more reliable outputs. For this reason, for RAG; We can say that it is a framework that provides functionality to improve the performance of large language models. Thanks to this flexible and powerful framework, RAG has achieved significant growth in the applications of large language models in the corporate field in just 3 years. RAG combines the power of large language models with structured and unstructured data, making enterprise information access more effective and faster than ever before. In this way, more competent and successful artificial intelligence services can be produced without the need for data science processes such as data preparation required by traditional virtual assistants, while spending minimum labor.

Working of RAG:

Step 1: Data collection

You must first gather all the data that is needed for your application. In the case of a customer support chatbot for an electronics company, this can include user manuals, a product database, and a list of FAQs.

Step 2: Data chunking

Data chunking is the process of breaking your data down into smaller, more manageable pieces. For instance, if you have a lengthy 100-page user manual, you might break it down into different sections, each potentially answering different customer questions.

This way, each chunk of data is focused on a specific topic. When a piece of information is retrieved from the source dataset, it is more likely to be directly applicable to the user’s query, since we avoid including irrelevant information from entire documents.

This also improves efficiency, since the system can quickly obtain the most relevant pieces of information instead of processing entire documents.

Step 3: Document embeddings

Now that the source data has been broken down into smaller parts, it needs to be converted into a vector representation. This involves transforming text data into embeddings, which are numeric representations that capture the semantic meaning behind text.

In simple words, document embeddings allow the system to understand user queries and match them with relevant information in the source dataset based on the meaning of the text, instead of a simple word-to-word comparison. This method ensures that the responses are relevant and aligned with the user’s query.

If you’d like to learn more about how text data is converted into vector representations, we recommend exploring our tutorial on [text embeddings with the OpenAI API](https://www.datacamp.com/tutorial/introduction-to-text-embeddings-with-the-open-ai-api).

Step 4: Handling user queries

When a user query enters the system, it must also be converted into an embedding or vector representation. The same model must be used for both the document and query embedding to ensure uniformity between the two.

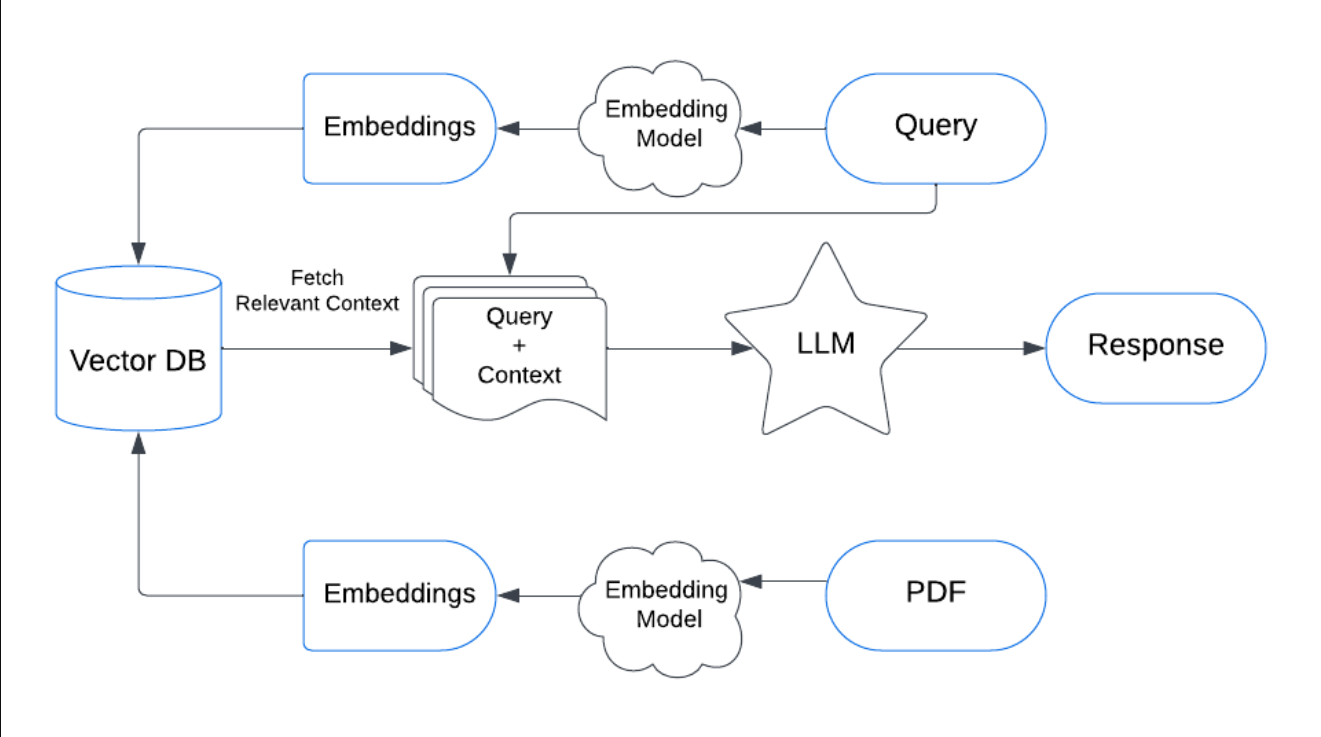
Once the query is converted into an embedding, the system compares the query embedding with the document embeddings. It identifies and retrieves chunks whose embeddings are most similar to the query embedding, using measures such as cosine similarity and Euclidean distance.

These chunks are considered to be the most relevant to the user’s query.

Step 5: Generating responses with an LLM

The retrieved text chunks, along with the initial user query, are fed into a language model. The algorithm will use this information to generate a coherent response to the user’s questions through a chat interface.

Flow chart:



The framework of RAG according to the above flow chart is

* Query Input: User input is passed into the system.
* Embedding Model: Converts the input (e.g., query or PDF) into vector representations.
* Vector Database (Vector DB): Stores embeddings and retrieves relevant contexts based on similarity search.
* LLM (Large Language Model): Processes the query and the fetched context to generate responses. Response Output: The generated response is provided to the user.

The embedding model our system employs is **Sentence Transformers**  and the vector database we emply is **FAISS.**

**Development Tools:**

Visual Studio Code for coding and debugging.

Streamlit interface for initial testing.

GOOGLE Gemini AI Studio of LLM API key and integration.

Proxy Local Host Live Server for Projecct View.

**Programming Language:**

AI Bot: Python Language with Langchain Library , Sentence Transformers from Hugging Face.

Web Page: HTML,CSS, JAVASCRIPT

FastAPI for clean integration of the BOT.

**Environment:**

All componets hosted locally in the host system environment . No cloud services are used .

**Summary:**

InfoBot provied easy summaries of the web page in a conversational format . This allows users to interact with the webpage directly . The retrival features leveraged by powerfull open-source LLMs provide precise and legit responses to asked questions. The UI framework is kept simple for simple asthetic look on the website and also for easy user interaction .This project aims to provide easy solution to developers and organizations to make their websites easy to navigate and provide simple effective user interactions.

**Features:**

1)Powerful Gemini LLM: This open sourced google llm is considerd to be one of the powerful AI models avilable in markert. This is used for generating user understandable conversational responses.

2)Langchain: This Library provides various user friendly features that can be added into the chatbot with minimal hiccups.

3)Precise Embeddings Generation: Sentence Transformers are used to generate precise embeddings.

3)Simple UI:The designed UI can be integrated into any traditional frameworked website in matter of lines for wasy integrations.

**Challenges Encountered:**

* Precise Responses: The responses generated had to be as legit and similar to as provided in the website. Extra and deficit information problems were solved in the process of building this model.
* Testing and Debbuging: Addressed multiple bugs during the testing phase, particularly related to the chatbot’s recommendation logic.

**LIMITATIONS:**

* Depends on the website administrators on how this bot is implemeted.
* If the quality and quantity of data is low , the quality responses generated by the bot is also drops
* The current framework is designed for HTML,CSS and JAVASCRIPT. Websites using Framewroks of Flask , Django or React or Vue might face integration challenges.

**Requirements**

**Non Functional Requirements:**

Performance

- The system should maintain responsiveness even under high traffic conditions.

- Chatbot responses must be delivered in under two seconds for optimal user experience.

Scalability

- Flexible architecture to support future enhancements and increased user interactions.

Usability

- A user-friendly interface that is easy to navigate and understand.

- Support for multi-device access, including desktops, tablets, and smartphones.

**Software Requirements:**

Python Libraries

The following libraries must be installed:

* FastAPI: For building the API.

Install via pip install fastapi.

* Uvicorn: ASGI server to run the FastAPI application.

Install via pip install uvicorn[standard].

* PyPDF2: For extracting text from PDFs.

Install via pip install PyPDF2.

* LangChain: For building the question-answering chain.

Install via pip install langchain.

* LangChain-Community FAISS: To store and retrieve text embeddings.

Install via pip install langchain-community.

* LangChain-Google-GenAI: Integration with Google Generative AI.

Install via pip install langchain-google-genai.

* Google Generative AI SDK (google.generativeai): To interact with the Generative AI API.

Install via pip install google-generativeai.

* python-dotenv: For loading environment variables from .env file.

Install via pip install python-dotenv.

External Services and Configuration

* Google Generative AI API:

A valid API key from Google Generative AI is required.

Set up the environment variable GOOGLE\_API\_KEY in a .env file in your project directory.

* FAISS (Facebook AI Similarity Search):

No external configuration is needed; the library handles everything in memory.

Development and Deployment

* Development Server:

Use Uvicorn for running the application:

bash

Copy code

uvicorn main:app --reload

* Production Deployment:

Use a production-ready server like gunicorn with uvicorn.workers.UvicornWorker.

**SYSTEM DESIGN**:

* Query Processing:

The user query is converted into vector embeddings using an Embedding Model.

These embeddings are used to perform similarity-based retrieval.

* Document Retrieval:

A Vector Database stores precomputed embeddings of documents or other data sources.

The query embeddings are matched against the stored embeddings to fetch relevant documents or contexts.

* Augmentation:

The fetched documents or context are paired with the query for input into the generation phase.

* Generation:
* The combined Query + Context is passed to a Large Language Model (LLM), which generates a response.

The response is returned to the user.

* Additional Inputs:

Data like PDFs can also be processed by the Embedding Model and stored in the Vector Database for future queries.