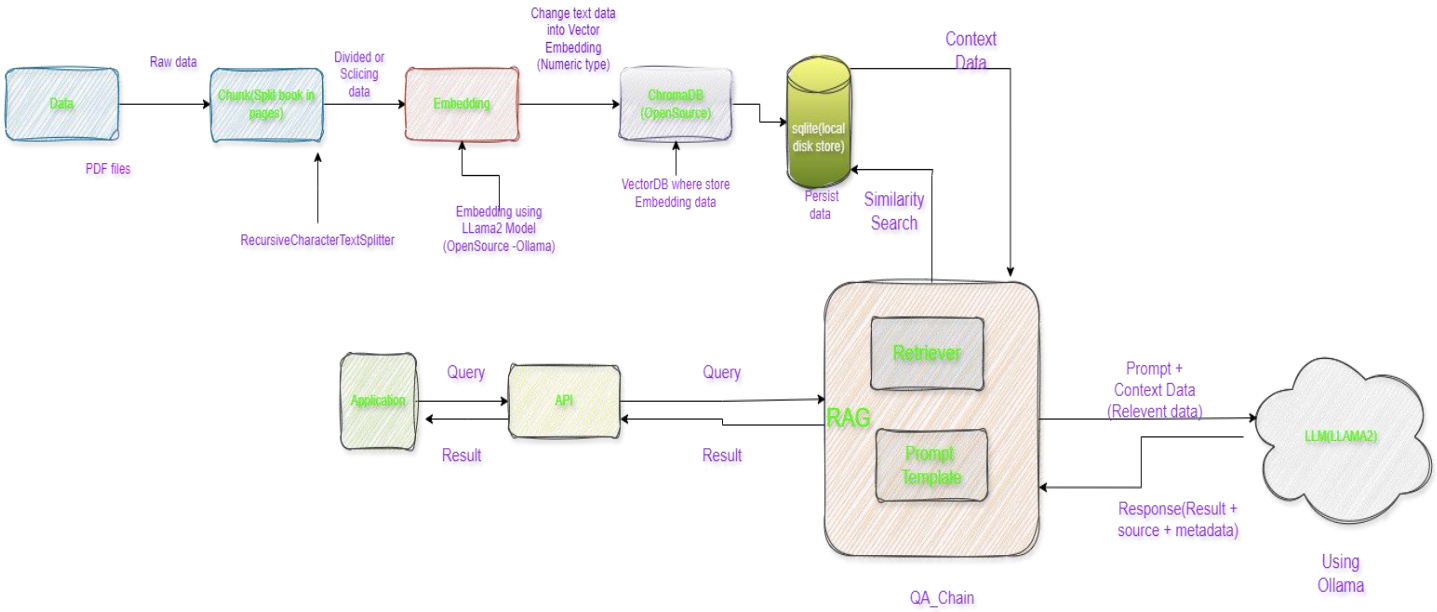
OVERVIEW OF THE CHATBOT ARCHITECTURE



Architecture of Chatbot for PDF

STEPS FOR BUILDING CHATBOT

I'd be glad to clarify the steps involved in creating a PDF chatbot using the provided using the provided code snippet:

1. **Getting And Loading the PDF File:**
   - You'll need a PDF document stored in your project's docs directory.
   - The code utilizes **langchain.** Directory to locate all PDF files within that directory using the **glob= './pdf'** pattern.
   - The **loader_cls=pypdfloader** argument specifies that the pypdfloader Class from the Langchain Library should be used for loading the PDF content.
2. **Splitting the PDF into Chunks:**
   - The **RecursiveCharacterTextSplitter** function takes the loaded PDF data and splits it into smaller, manageable segments.
   - This could involve dividing the document by pages, sections, or other logical groupings based on your specific needs**.**
3. **Embedding Text Chunks into Numerical Representation:**
   - The **OllamaEmbeddings(model='llama2')** function creates embeddings for each text chunk.

- Embeddings are essentially numerical representations that capture the meaning and context of the text. This allows the retrieval and question answering models to efficiently work with the document's content.
- The **llama2** model in this case suggests the use of a pre-trained language model (LLM) for generating these embeddings.

4. **Saving Embeddings in a Vector Database:**
   - The **Chromadb(OpenSource VectorDatabase** function is used to create a vector database **(ChromaDB)** to store the generated embeddings. This enables efficient retrieval and searching of similar content later on.
   - **Chroma.from_documents** assembles the embeddings and corresponding text chunks into a document format for storage.
   - **persist_directory** specifies the directory where the database will be stored on your local disk.
   - **vectordb.persist()** permanently saves the database contents.
5. **Creating Retrievers and LLM Model:**
   - A retriever object is created for each text chunk. Retrievers are responsible for finding relevant information from the corpus (in this case, the embeddings database) based on a given query.
   - The **retriever.as_retriever()** function likely prepares the retriever object for use in the question answering system.
   - LLM **model Ollama(base_url='http:/localhost:11434',model="llama2")** indicates the use of the **Ollama** LLM, probably running locally at http://localhost:11434. The **model="llama2"** part specifies the specific LLM variant employed.
6. **Building the Retrieval-Augmented Generation(RAG) Chain:**
   - The from **langchain.chains** import **RetrievalQA** function imports the **RetrievalQA** class, which is designed for building RAG chains.
   - RAG **(Retrieval-Augmented Generation)** combines retrieval from a knowledge base **(here, the ChromaDB)** with LLM generation to provide more informative and contextually relevant answers.
   - **qa_gen=RetrievalQA.from_chain_type(llm=llm_llama, chain_type="stuff", retriever=retriever, return_source_documents=True)** creates a **qa_gen** object that embodies the RAG chain.

- **llm=llm_llama** assigns the previously defined LLM model to the chain.
  chain_type="stuff" (the exact meaning depends on the LangChain library implementation) likely indicates the type of chain being created (possibly a generic retrieval and generation chain).
- **retriever=retriever** specifies the retriever object to be used within the chain.
- **return_source_documents=True** instructs the chain to return the source documents (text chunks) that were most relevant to the query in the response.

7. **Interacting with the Chatbot (Querying):**
   - Once the qa_gen object is constructed, you can interact with the chatbot by providing a question.
   - **qa_gen(question)['result']** extracts the answer generated by the RAG chain for the given question. This answer would likely combine information retrieved from the **ChromaDB** embeddings with text generation from the LLM model.
   - The **return_source_documents=True** setting in the **qa_gen** creation might also provide details about the relevant text chunks that contributed to the answer.

EXPLANATION OF HOW RAG, VECTORDB, EMBEDDING, AND LLM FRAMEWORKS ARE UTILIZED

RAG(Retriever Augmented Generation)

General-Purpose language models can be fine-tuned to achieve several common tasks such as sentiment analysis and named entity recognition. These tasks generally don't require additional background knowledge. For more complex and knowledge-intensive task, it's possible to build a language model-based system that accesses external knowledge sources to complete tasks. This enables more factual consistency, improves reliability of the generated responses, and help to mitigate the problem of "hallucination"

Meta AI researchers introduced a method called Retrieval Augmented Generation(RAG) to address such knowledge-intensive tasks. RAG combines an information retrieval component with a text generator model. RAG can be fine-tuned and its internal knowledge can be modified in an efficient manner and without needing retraining of the entire model. RAG takes an input and retrieves a set of relevant/supporting documents given a source(e.g Wikipedia). The documents are concatenated as context with the original input prompt and fed to the text generator which produces the final output. This makes RAG adaptive for situations where facts could evolve overtime. This is very useful as LLM's parametric knowledge is static. RAG allows language models to bypass retraining, enabling access to the latest information for generating reliable outputs via retrieval-based generation.

Large Language models (LLMs) have demonstrated an impressive ability to store and deploy vast knowledge in response to user queries while this has enabled the creation of powerful AI systems like ChatGPT, compressing world knowledge in this way has **two limitations**

First, an LLM's knowledge is static, i.e., not updated as new information becomes available.

Second, LLMs may have an insufficient "understanding" of niche and specialized information that was not prominent in their training data. These limitations can result in undesirable (and even fictional model responses to user queries.

Retrieval Augmented Generation (RAG) has gained popularity due to its ease of implementation and its ability to address challenges encountered by large language models (LLMs), such as context loss and producing inaccurate information, often referred to as 'hallucinations'. RAG enhances prompts by supplementing them reliable data, ensuring that LLM responses are accurate. Users can easily prompt the LLM to provide quick and pertinent answers by storing organizational documents in a vector database.

RAG consists of Two distinct phases: retrieval and content generation. In the retrieval phase, algorithms search for and retrieve relevant information from external knowledge base. This information is then used in the generative phase, where the LLM synthesizes an answer based on both the augmented prompt and its internal representation of training data.

Phase 1: Retrieval

- Relevant information is retrieved from external sources based on the User's prompt or question .
- Sources vary depending on the context (open-domain internet vs closed domain enterprise data).

Phase 2: Content Generation

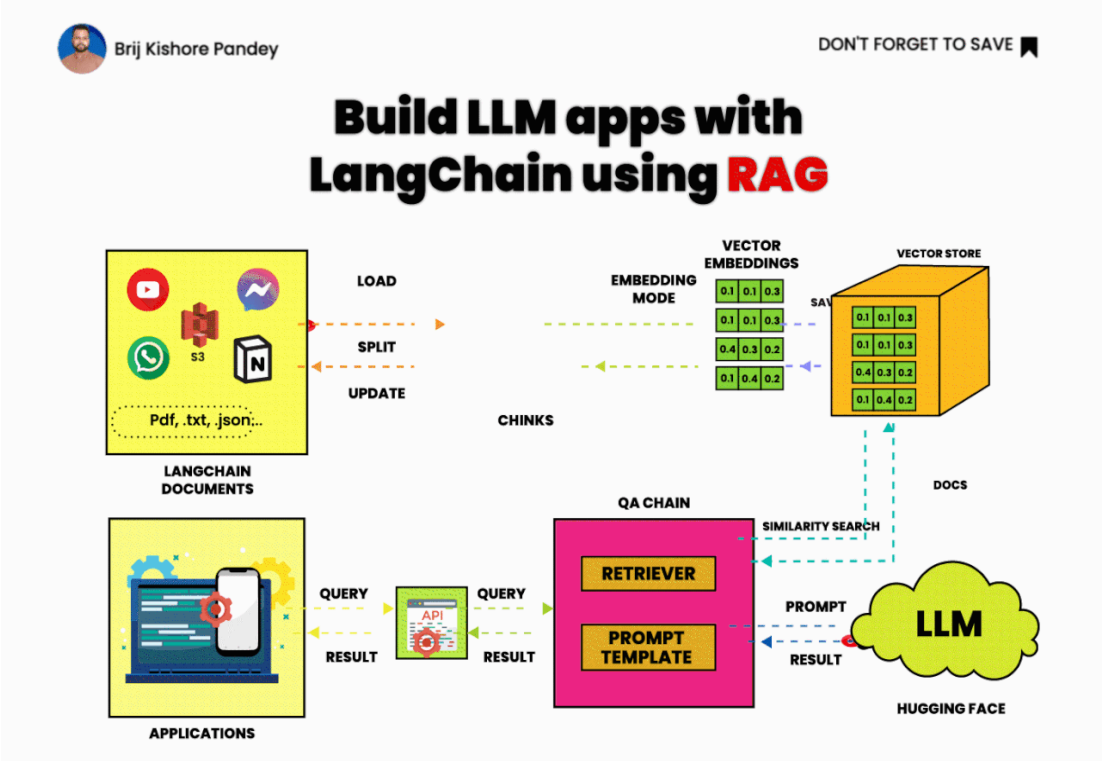
- The retrieved information is appended to the user's prompt and fed to the LLM.
- The LLM generates a personalized answer based on the augmented prompt and its internal knowledge base.
- The answer can be delivered with links to its sources for transparency.

The RAG Workflow

1. Setting up a knowledge Base

   Prepare a unified data repository by ingesting information from diverse, formatting it into a searchable 'Document' object Representation.
2. Data Handling
   - Set up Vector Database with efficient indexing.
   - Extract data from documents.
   - Divide documents into chunks.
   - Transform chunks into embeddings.
3. Retrieval
   - Once the user enters a query, get the embeddings from user queries and search the vector database to fetch the relevant embedding from the chunks.
4. Augmentation
   - Retrieved document embeddings are combined with the original query given by user to from a prompt. This Prompt includes both the query and the relevant context. This step ensures a comprehensive understanding of the user's input.
5. Generation

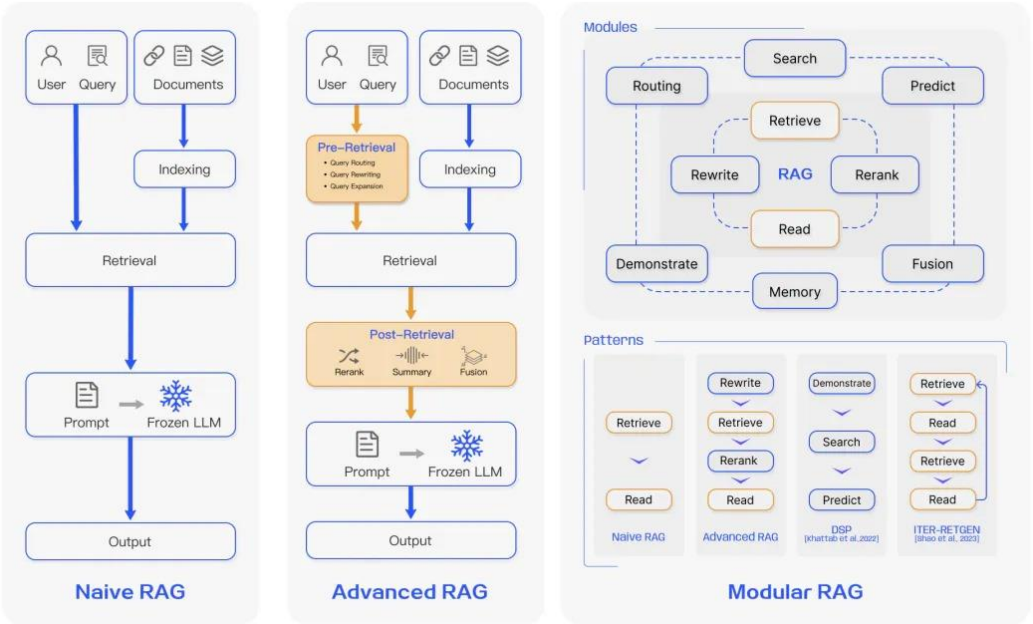
   With all this information, Prompt is given to LLM which uses context from retrieved documents to generate an accurate, informative, and contextually relevant response.



RAG Workflow

Types of RAG:

1. **Naïve RAG**
2. **Modular RAG**
3. **Advance RAG**



Types of RAG

1. **Naïve RAG: -**

   Naïve RAG is essentially a Retrieve=> Read approach which focussed on retrieving information and comprehending it.

   At its most basic, Retrieval Augmented Generation can be summarized in three steps-

   1. **Indexing** of the **documents**
   2. **Retrieval** of the context with respect to an input query
   3. **Generation** of the response using the input query and retrieved context.

Challenges in Naïve RAG

Retrieval Quality:

- **Low Precision** leading to Hallucinations/Mid-air drops
- **Low Recall** resulting in missing relevant info
- **Outdated information**

Augmentation:

- **Redundancy and Repetition** when multiple retrieved documents have similar information
- **Context Length** Challenges

Generation Quality:

- Generations are not **Grounded** in the context.
- Potential of **toxicity and bias** in the response
- **Excessive dependence** on augmented context

2. Advanced RAG:

   Advanced RAG is adds to the Retrieve => Read approach by adding it into a Rewrite and Rerank components to improve relevance and groundedness

   To address the inefficiencies of the Naïve RAG approach, Advanced RAG approaches implement strategies focussed on three process-

   Pre-Retrieval Retrieval Post Retrieval

   Advanced RAG Concepts

   Pre-Retrieval/Retrieval Stage

   - Chunk Optimization
   - Metadata Integration

- Indexing Structure
- Query Rewriting
- Fine-tuned Embeddings
- Etc

Post Retrieval Stage

- Information Compression
- Reranking

3. Modular RAG: takes everything a notch by Providing flexibility and adding modules like Search, Routing etc

   The SOTA in Retrieval Augmented Generation is a modular approach which allows components like Search, Memory, and reranking modules to be configured

Some RAG Modules:

- Search
- Memory
- Fusion
- Extra Generation
- Task Adaptable Module

Benefits of RAG:

1. Providing up-to-date and accurate Response
2. Reducing inaccurate responses, or hallucinations
3. Providing domain-specific, relevant responses
4. Being efficient and cost-effective

VectorDB

A vector database is a database that stores information as vectors, which are numerical representations of data objects, also known as vector embeddings. It leverages the power of these vector embeddings to index and search across a massive dataset of unstructured data and semi-structured data, such as images, text, or sensor data. Vector databases are built to manage vector embeddings, and therefore offer a complete solution for the management of unstructured and semi-structured data. A vector database is different from a vector search library or vector index: it is a data management solution that enables metadata storage and filtering, is scalable, allows for dynamic data changes, performs backups, and offers security features. Vector embeddings are numerical representations of high-dimensional data such as text, images, audio, and video. They are created using mathematical algorithms that

transform the raw data into vectors with a fixed number of dimensions, ranging from tens to thousands depending on the complexity of the data.

Vector embeddings are used to capture the semantic or contextual meaning of the data, allowing it to be compared and analysed in a meaningful way. For example, in natural language processing, words can be represented as vectors in a high-dimensional space, where words with similar meanings are closer together in the space than words with different meanings.

A traditional database stores information in tabular form, and indexes data by assigning values to data points. When queried, a traditional database will return results that exactly match the query.

A vector database stores vectors in form of embeddings and enables vector search, which returns query results based on similarity metrics(rather than exact matches). A vector database "steps up" where a traditional database cannot: it is intentionally designed to operate with vector embeddings.

A vector database is a specific kind of database that saves information in the form of multi-dimensional vectors representing certain characteristics or qualities.

The number of dimensions in each vector can vary widely, from just a few to several thousand, based on the data's intricacy and detail. This data, which could include text, images, audio, and video, is transformed into vectors using various processes like machine learning models, word embeddings, or feature extraction techniques.

The primary benefit of a vector database is its ability to swiftly and precisely locate and retrieve data according to their vector proximity or resemblance. This allows for searches rooted in semantic or contextual relevance rather than relying solely on exact matches or set criteria as with conventional databases.

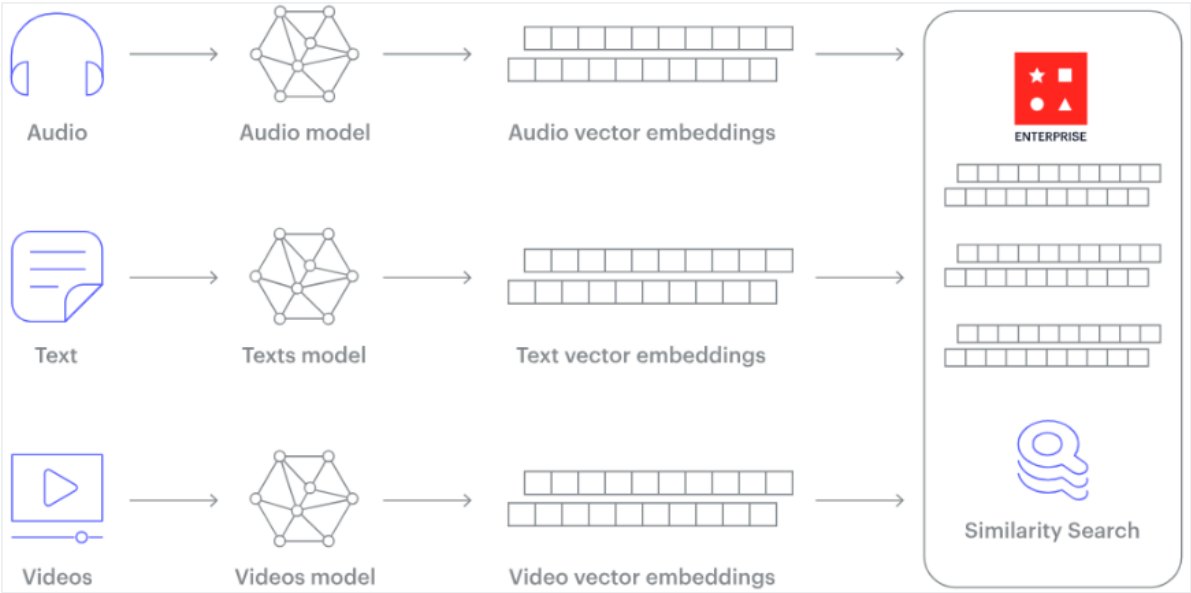
For instance, with a vector database, you can:

- Search for songs that resonate with a particular tune based on melody and rhythm.
- Discover articles that align with another specific article in theme and perspective.
- Identify gadgets that mirror the characteristics and reviews of a certain device.

Types of VectorDB

- **ChromaDB(Open Source)**
- **Pinecone (Source available or commercial)**
- **Weaviate (Source available or commercial)**
- **LanceDB(Open Source)**

Working of Vector Database



Working of Vector Database

Unstructured data, such as text, images, and audio, lacks a predefined format, posing challenges for traditional databases. To leverage this data in artificial intelligence and machine learning applications, it's transformed into numerical representations using embeddings.

Embedding is like giving each item, whether it's a word, image, or something else, a unique code that captures its meaning or essence. This code helps computers understand and compare these items in a more efficient and meaningful way. Think of it as turning a complicated book into a short summary that still captures the main points.

This embedding process is typically achieved using a special kind of neural network designed for the task. For example, word embeddings convert words into vectors in such a way that words with similar meanings are closer in the vector space.

This transformation allows algorithms to understand relationships and similarities between items.

Essentially, embeddings serve as a bridge, converting non-numeric data into a form that machine learning models can work with, enabling them to discern patterns and relationships in the data more effectively.

EMBEDDING

Embeddings are made by assigning each item from incoming data to a dense vector in a high-dimensional space. Since close vectors are similar by construction, embeddings can be used to find similar items or to understand the context or intent of the data. While LLMs are primarily concerned with language, similar GenAI models can serve purposes such as text-to-image, or

audio-to-text, among others. These models can transform data between multiple modalities, including text, images, video, and audio. Regardless of purpose, each model will interpret the underlying meaning of its input and, using embeddings as an intermediary, generate the most likely output according to the training data.

Generating value from embeddings

Now that you understand what embeddings are and how you can transform your data into them, let's walk through an example that shows how you can index a knowledge base of company information into one searchable, queryable engine.

Identifying company similarity

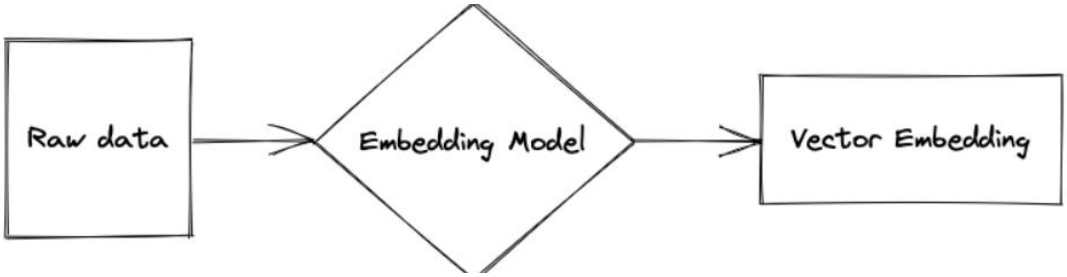
In the context of a company similarity use-case, you can create a vector representation for each company based on its data. The GenAI step-change is that similarity is no-longer based on a simple keyword search, but instead on an ontological understanding where similar items are close together in the embedding space. The embeddings themselves can be generated from each companies' unstructured text data passed through an LLM. These embeddings alongside traditional structured data can create a complete picture of a company, and allow us to conduct more intelligent searches across your knowledge base. This enables a more accurate and intuitive understanding of similarity, across languages, regions and industries.

Intelligent search

To leverage the power of your embeddings, you can use them to conduct similarity searches across unstructured text data. Input queries can be transformed using the same transformer model used to index the data followed by the similarity measure the model was trained on,

Embracing embeddings

There are many ways you can get started with embeddings and transformers. There are courses available on transformers and the architecture behind encoders and decoders which can help you understand the fundamentals of this technology and begin developing solutions with it. Learning about foundation models and how they can be used to fine tune to a certain situation will then be a good next step.



Embedding Raw Data

LLM Framework

Orchestration Framework in Action. Orchestration is the process of managing and controlling Large language models (LLM) in a way that optimizes their performance and effectiveness. This includes tasks such as

- Prompting LLMs: Generating Effective prompts that provide LLM's with the proper context and information to produce desired outputs
- Chaining LLMs : Combining the outputs of multiple LLMs to achieve more complex or nuanced results.
- Managing LLM resources: Efficiently allocating and managing LLM resources to meet the demands of an application.
- Monitoring LLM Performance: Tracking metrics to identify and address issues.

LLM orchestration is becoming increasingly crucial as LLMs are used in a border range of applications, such as natural language generation, machine translation, and question-

answering. By effectively orchestrating LLMs developers can build more robust and reliable applications.

Several different LLM Orchestration frameworks are available, each with strengths and weaknesses. Some popular frameworks include:

- LangChain: A framework that provides a high-level API for interacting With LLMs
- Lamaindex: a Framework allowing developers to query their private data using LLMs.
- Orkes: A framework that provides a workflow engine for building complex LLM application.

LLM orchestration frameworks provide a high-level interface for managing and controlling large language models (LLMs). They abstract away the complexities of prompt generation, resource management, and performance monitoring to enable developers to interact with LLMs easily. LLM orchestration frameworks can significantly improve developer productivity and application performance by streamlining the development process.

Here are some of the most popular LLM orchestration frameworks:

LangChain: LangChain is a Python-based framework that provides a declarative API for defining LLM workflows. It allows developers to compose complex sequences of LLM interactions, including prompting, chaining, and conditional branching. LangChain also offers features for managing LLM resources and monitoring performance.

1. **LangChain**

LangChain is an open-source framework for developing applications powered by large language models (LLMs). It provides a high-level API that abstracts away the details of working with LLMs, making it easier for developers to build and deploy complex applications. LangChain is also modular and extensible, allowing developers to customize it to their needs.

Benefits of using LangChain

There are many benefits to using LangChain, including:

1. **Increased developer productivity:** LangChain makes it easier for developers to build and deploy LLM-powered applications by abstracting away the details of working with LLMs.
2. **Improved application performance:** LangChain can help to improve application performance by optimizing the use of LLM resources.

3. **Reduced development costs:** LangChain can help reduce development costs by making building and maintaining LLM-based applications easier.
4. **Increased scalability and reliability of LLM-based applications:** LangChain can help to make LLM-based applications more scalable and reliable by providing a way to manage and control LLMs across multiple nodes or machines.

Use cases of LangChain

LangChain can be used for a wide variety of applications, including:

1. **Natural language generation:** LangChain can generate text, such as blog posts, articles, and marketing copy.
2. **Machine translation:** LangChain can translate text from one language to another.
3. **Question answering:** LangChain can be used to answer questions comprehensively and informally.
4. **Chatbots:** LangChain can be used to build chatbots that engage in natural and meaningful conversations with users.

Getting started with LangChain

LangChain is easy to get started with. Several tutorials and documentation are available online to help, and pre-built applications can also be used as a starting point.

The future of LangChain

LangChain is a rapidly evolving framework. The developers are constantly adding new features and capabilities. As LLMs grow, LangChain will become even more critical for building robust and reliable applications.

LangChain is a powerful and versatile framework for building various LLM-powered applications. It is easy to use and can help improve developer productivity, application performance, and development costs. If you want a framework to help you build LLM-powered applications, I highly recommend LangChain.

INSTRUCTIONS FOR SETTING UP THE ENVIRONMENT AND RUNNING THE CHATBOT

Instruction Setting Up Environment

Before starting, make sure you have Python 3.8+ installed. To run the Q&A Chatbot, follow these steps:

1. Create Conda Env:
   conda create --name pdf_env python
2. Clone This Repository.
   git clone https://github.com/GENRATECODE/pdf_chatbot.git
   cd pdf_chatbot
3. Install the Necessary Libraries.
   pip install -r requirement.txt
4. Install Ollama for Mac/Windows/Linux.
   https://ollama.com/download
5. Download LLM model llama2 on Local Own System
   Ollama run llama2

Issuse During Installation Chromadb

```
uilding wheel for hnswlib (pyproject.toml) ... error
  error: subprocess-exited-with-error

  × Building wheel for hnswlib (pyproject.toml) did not run successfully.
  │ exit code: 1
  ╰─> [21 lines of output]
      running bdist_wheel
      running build
      running build_ext
      creating tmp
      x86_64-linux-gnu-gcc -pthread -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O2 -Wall -g -fstack-protector-strong -Wformat -Werror=format-security -g -fwrapv -O2 -fPIC -I/home/namanc/codes/langchian_chromadb/venv/include -I/usr/include/python3.10 -c /tmp/tmp822lkln8.cpp -o tmp/tmp822lkln8.o -std=c++14
```

```
    x86_64-linux-gnu-gcc -pthread -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O2 -Wall -g -fstack-protector-strong -Wformat -Werror=format-security -g -fwrapv -O2 -fPIC -I/home/namanc/codes/langchian_chromadb/venv/include -I/usr/include/python3.10 -c /tmp/tmpw33wg22s.cpp -o tmp/tmpw33wg22s.o -fvisibility=hidden
    building 'hnswlib' extension
    creating build
    creating build/temp.linux-x86_64-cpython-310
    creating build/temp.linux-x86_64-cpython-310/python_bindings
    x86_64-linux-gnu-gcc -pthread -Wno-unused-result -Wsign-compare -DNDEBUG -g -fwrapv -O2 -Wall -g -fstack-protector-strong -Wformat -Werror=format-security -g -fwrapv -O2 -fPIC -I/tmp/pip-build-env-izlfj5h4/overlay/lib/python3.10/site-packages/pybind11/include -I/tmp/pip-build-env-izlfj5h4/overlay/lib/python3.10/site-packages/numpy/core/include -I./hnswlib/ -I/home/namanc/codes/langchian_chromadb/venv/include -I/usr/include/python3.10 -c ./python_bindings/bindings.cpp -o build/temp.linux-x86_64-cpython-310/./python_bindings/bindings.o -O3 -fopenmp -DVERSION_INFO=\"0.7.0\" -std=c++14 -fvisibility=hidden
    In file included from /tmp/pip-build-env-izlfj5h4/overlay/lib/python3.10/site-packages/pybind11/include/pybind11/detail/../attr.h:13,
                     from /tmp/pip-build-env-izlfj5h4/overlay/lib/python3.10/site-packages/pybind11/include/pybind11/detail/class.h:12,
                     from /tmp/pip-build-env-izlfj5h4/overlay/lib/python3.10/site-packages/pybind11/include/pybind11/pybind11.h:13,
                     from /tmp/pip-build-env-izlfj5h4/overlay/lib/python3.10/site-packages/pybind11/include/pybind11/functional.h:12,
                     from ./python_bindings/bindings.cpp:2:
    /tmp/pip-build-env-izlfj5h4/overlay/lib/python3.10/site-packages/pybind11/include/pybind11/detail/../detail/common.h:266:10: fatal error: Python.h: No such file or directory
      266 | #include <Python.h>
          |          ^~~~~~~~~~
    compilation terminated.
```

```
    error: command '/usr/bin/x86_64-linux-gnu-gcc' failed with exit code 1
    [end of output]

  note: This error originates from a subprocess, and is likely not a problem with pip.
  ERROR: Failed building wheel for hnswlib
Failed to build hnswlib
ERROR: Could not build wheels for hnswlib, which is required to install pyproject.toml-based projects
```

Resolve this error in windows install
https://visualstudio.microsoft.com/visual-cpp-build-tools/

install this package from this link
more detail search
https://stackoverflow.com/questions/76592197/python-installation-of-chromadb-failing

Run of Chatbot:

We use flet (flutter in python)library for GUI Backend connected.
Run the application from your terminal with
flet chatbot_ui.py
or
python chatbot_ui.py

If you want customization save pdf file in docs file then run this command on the terminal
Save file location **docs/python1.pdf** store pdf in docs folder then run
vectordb_embedding.py file
python vectorDB_Embedding.py

EVALUATION CRITERIA

Langchain and LLamaIndex have made it quite simple. Developing highly impressive Large language Model (LLM) applications is achievable through brief training and verification on a limited set of examples. However, to enhance its robustness, thorough testing on a dataset that accurately mirrors the production distribution is imperative.

Note: RAG is a great tool to address hallucination is LLMs but even RAGs can suffer from hallucinations.

This can be because –

- The retriever fails to retrieve relevant context or retrieves irrelevant context
- The LLM, despite being provided the context, does not consider it
- The LLM instead of answering the Query picks irrelevant information from the context

Two processes, therefore, to focus on from an evaluation perspective –

Search & Retrieval

- How good is the retrieval of the context from the vector Database?
- Is it relevant to the query?
- How much noise (irrelevant information) is present?

Generation:

- How good is the generated response?
- Is the response grounded in the provided context?
- Is the response relevant to the query?

Evaluation Data:

To evaluate RAG pipelines, the following four data points are recommended

- A set of Queries or Prompts for evaluation
- Retrieved Context for each Prompt
- Corresponding Response or Answer from LLM
- Ground Truth or Known correct response

Evaluation Metrics:

Evaluating Generation

Faithfulness ----- Is the Response faithful to the Retrieved Context?

Answer Relevance ---- Is the Response relevant to the Prompt?

Retrieval Evaluation

Context Relevance --- Is the Retrieved Context relevant to the Prompt ?

Context Recall ---- Is the Retrieved Context aligned to the Ground Truth?

Context Precision ---- Is the Retrieved Context ordered Correctly?

Overall Evaluation

Answer Semantic Similarity

Is the Response semantically Similar to the **Ground Truth**?

Answer Correctness

Is the Response semantically and factually similar to the **Ground Truth**?

Code Link: - https://github.com/GENRATECODE/pdf_chatbot