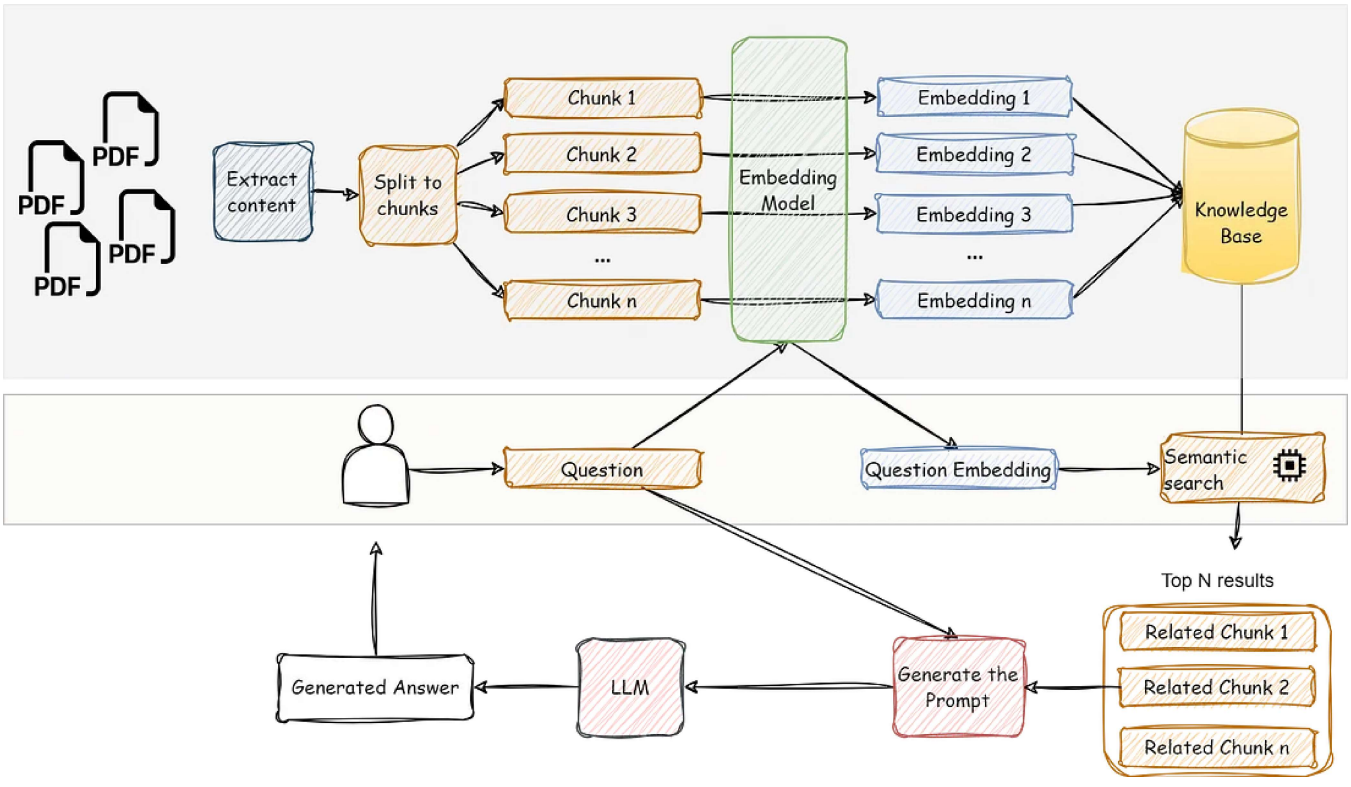
Bring Your Data to Life: Creating a Chatbot with LLM, LangChain, Vector DB (Locally on Docker), and Streamlit

mutaz younes · Follow
9 min read · Sep 7, 2023



Introduction:

Hey there, fellow data explorer! 🚀 Ever found yourself lost in a sea of data and wished you had a trusty sidekick to help you navigate it all? Well, buckle up, because we're about to dive headfirst into the fascinating world of conversational AI, where data meets chatbots, and it's a whole lot of fun!

In this article, we're going to take you on a journey to create a chatbot that's not just smart but whip-smart. Picture having a chatbot that doesn't just hear your questions but answers them with real insight. Sounds too good to be true? Well, it's not! We're going to tap into the incredible powers of Language Model (LLM) technology, mak the most of a nifty vector database, and wrap it all up with a user-friendly interface thanks to Streamlit.

And here's the kicker — we're not just going to give you the rundown; we're going to roll up our sleeves and code it together. So, dust off your coding chops, because it's about to get hands-on in here! By the time we're done, you'll have the know-how to create your very own data-savvy chatbot. Ready to breathe life into your data? Let's dive in and make data conversations the coolest thing since sliced bread!

The Final product:

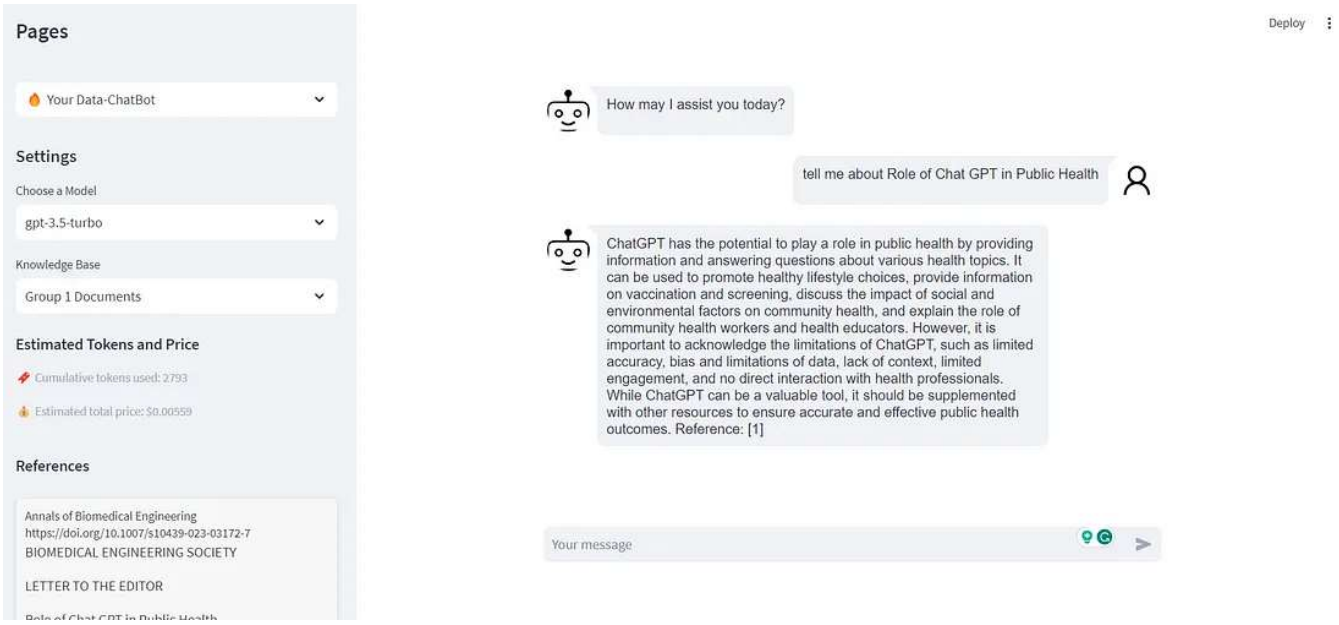


Table of Contents

- Vector DB (nucliaDB)
- Setting the DataBase (Docker)

3. The Pipeline

- Managing Files and Chunking
- Reading Files and Chunking
- Loading Documents
- Text Splitting
- Embeddings and Saving to Vector DB
- Iterating Through Files
- The above steps at once

4. User's Questions and Generating an Answer

- User Interaction in Streamlit
- Embedding User's Questions and Finding Related Chunks
- OpenAI Model Integration
- Generating the Answer

5. Conclusion

- Recap of the Journey
- Resources and GitHub Repository

Understanding the Tools (You can skip this part)

1. Streamlit (UI)

Streamlit is a Python library that simplifies the process of creating interactive web applications with minimal effort. In the context of your chatbot, Streamlit will serve as the user interface (UI), providing a seamless and intuitive way for users to interact with your data. In this section, we'll explore the fundamentals of Streamlit,

from setting up your environment to designing a user-friendly interface that facilitates natural conversations with your chatbot.

2. OpenAI API

The OpenAI API is the engine that powers the language understanding of your chatbot. It's your gateway to state-of-the-art language models like GPT-3, which can comprehend and generate human-like text. We'll delve into how to integrate the OpenAI API into your chatbot, set up authentication, and send queries to the model to receive meaningful responses. Understanding how to utilize this API effectively is essential for the success of your chatbot.

3. Vector DB

In the world of data management and retrieval, Vector Databases have emerged as a game-changing technology. These databases are designed to efficiently store and retrieve data represented as vectors, opening up new possibilities for various applications, including search engines, recommendation systems, and more.

One key advantage of Vector Databases is their ability to handle high-dimensional data with ease. Unlike traditional databases, which rely on structured tables, Vector Databases organize information in vector representations. This approach allows for flexible and scalable data storage, making it suitable for complex data types such as text, images, and embeddings.

Setting Up the Database with Docker

To unleash the full potential of NucliaDB as a code search engine, the first step is to set up your NucliaDB using Docker. If you haven't already, ensure you have Docker installed and ready to go. Follow these simple steps to get your NucliaDB up and running:

```
docker run -it \
   -e LOG=INFO \
   -p 8080:8080 \
   -p 8060:8060 \
   -p 8040:8040 \
   -v nucliadb-standalone:/data \
   nuclia/nucliadb:latest
```

With this Docker command, you'll initiate your local NucliaDB instance, complete with all the necessary configurations. Now, let's move on to preparing your data.

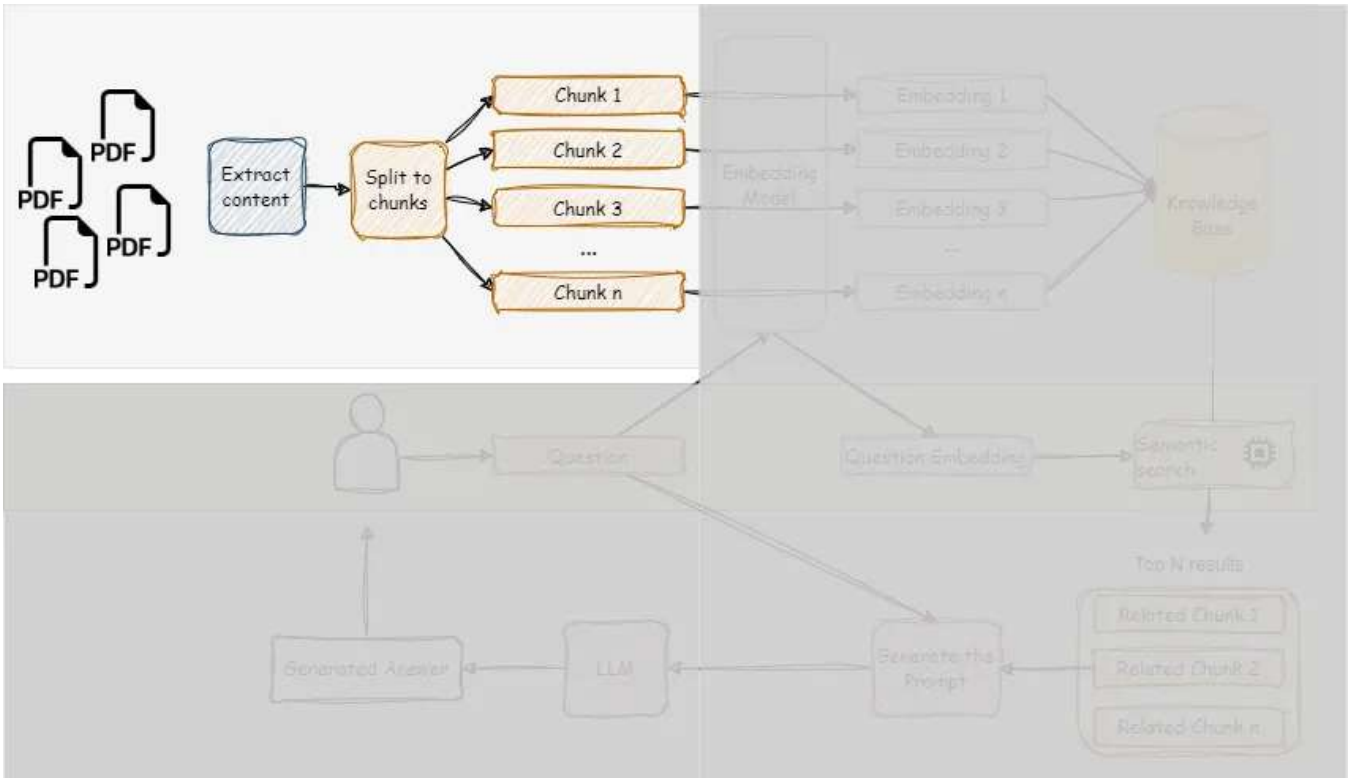
The Pipeline

Now the fun begins!!

1. Managing Files and Chunking

Let us break down the first image we had in this article, and understand how to implement each part of it.

Reading Files and Chunking



In the initial stages of creating your data-driven chatbot, one of the first challenges is dealing with the data itself. You might have a repository of documents, and processing them as a whole can be daunting. Here's a breakdown of how you're handling this in your code:

Loading Documents:

You start by using the `DirectoryLoader` from `langchain.document_loaders` to load documents from a specified directory. These documents likely contain the data you want to work within your chatbot.

```
loader = DirectoryLoader(directory)
documents = loader.load()
```

This step ensures that your code is aware of the documents you have and prepares them for further processing.

Text Splitting:

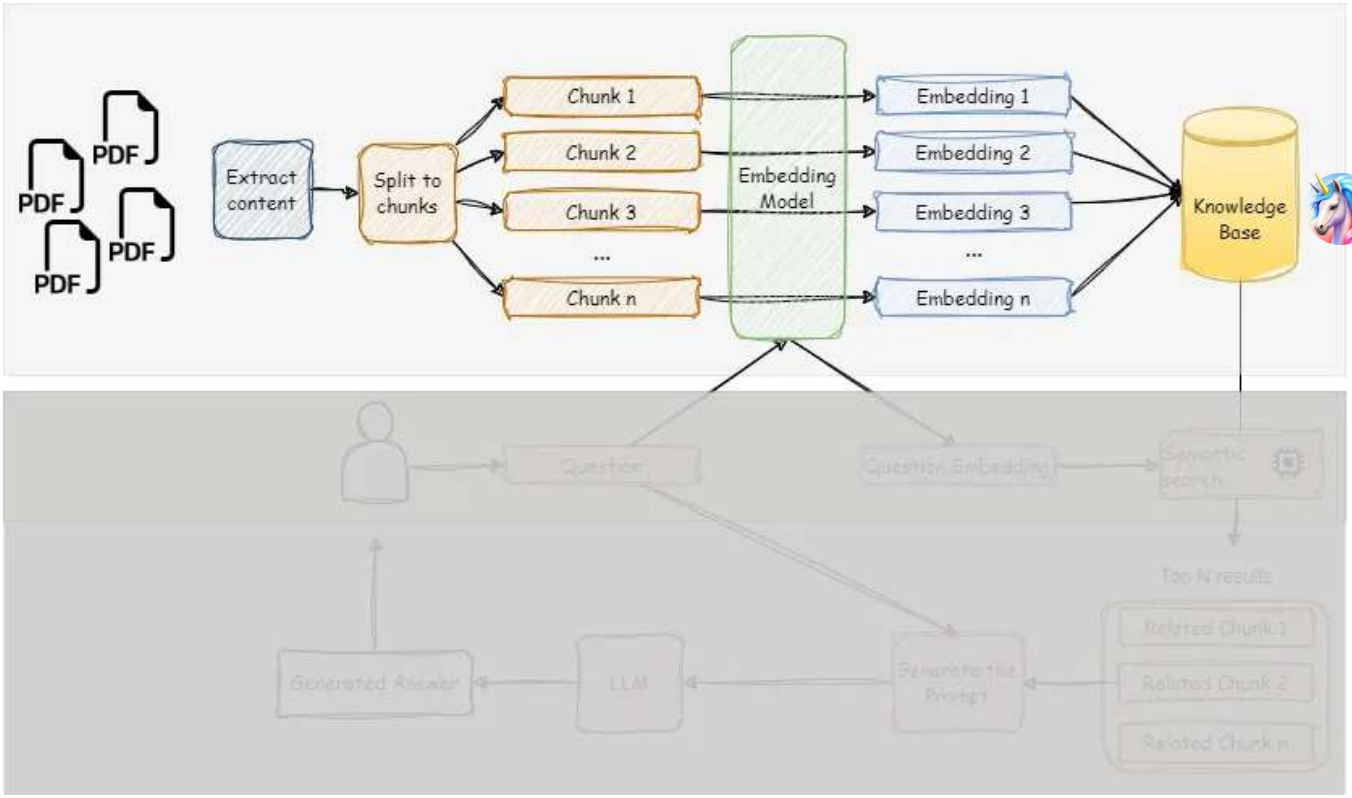
To make the data more manageable and suitable for processing, you're using a text splitter, specifically the `RecursiveCharacterTextSplitter`. This tool is crucial for breaking down lengthy documents into smaller, more digestible chunks

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size, chunk_ove
return text_splitter.split_documents(documents)
```

Here, `chunk_size` and `chunk_overlap` are parameters that define how you want to split your text. Chunks help in processing data step by step, making it easier to work with and reducing the computational load when dealing with large documents.

2. Embeddings and Saving to Vector DB

Embeddings are a fundamental concept in natural language processing (NLP). We'll dive into how embeddings are used to represent your data and queries in a format that LLMs can understand. Understanding embeddings is key to making your chatbot proficient at understanding and working with your data.



Iterating Through Files:

Once you have your documents split into chunks, your code proceeds to iterate through them. In the provided code, this is done within a `for` loop that goes through each file, creating chunks along the way.

```
for i, file in enumerate(tqdm(files, desc="Processing files")):
```

Inside this loop, you're processing each chunk of data, encoding it using the Sentence Transformer model, and uploading it to your knowledge base.

```
vectors = model_bge.encode([file.page_content])
my_kb.upload(
    key=f"mykey{i}",
    text=file.page_content,
    labels=[file.metadata["source"]],
    vectors={"bge": vectors[0]},
)
```

This step is where the magic happens. You're not only splitting the documents into manageable pieces but also preparing them for analysis and interactions with your

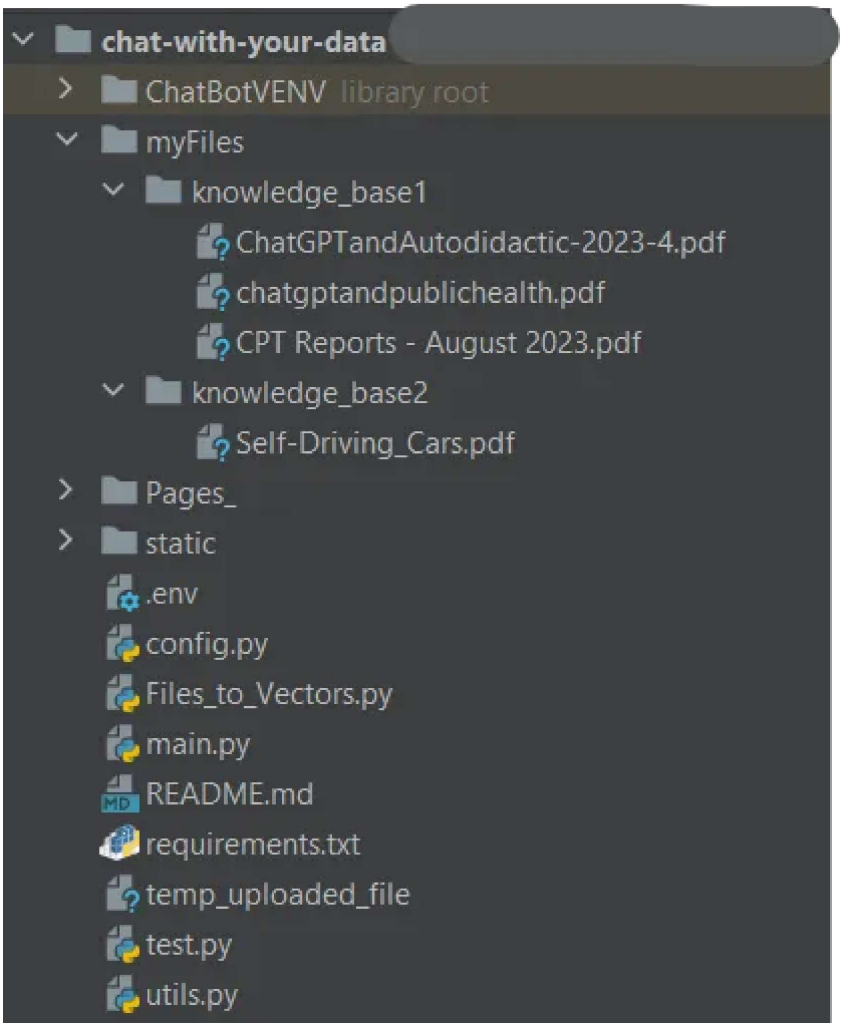
chatbot. The `model_bge` is used to convert text into meaningful numerical representations, making it easier for your chatbot to understand and respond to user queries.

In a nutshell, this part of your code takes your data, chops it into pieces, and transforms it into a format that's ready to be used by your chatbot. It's a crucial step in the data preparation pipeline, setting the stage for the more advanced NLP and AI processes that follow.

4. The above steps at once

Here is the entire code that will load your data, create chunks, generate the embeddings, and finally save them to the vector database.

Make sure you place your files in the myFiles folder.



Let's call this 'Files_to_vecotrs.py'

```python
# Import necessary modules and libraries
from langchain.document_loaders import DirectoryLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from nucliadb_sdk import create_knowledge_box, delete_kb
from sentence_transformers import SentenceTransformer
from tqdm import tqdm

# Function to split documents from a directory
def split_docs(directory, chunk_size=4300, chunk_overlap=300):
    """
    Load and split documents from the given directory.

    Args:
        directory (str): The path to the directory containing the documents.
        chunk_size (int): The size of document chunks to split.
        chunk_overlap (int): The overlap between document chunks.

    Returns:
        list: A list of document chunks.
    """
    # Initialize a directory loader
    loader = DirectoryLoader(directory)
    # Load documents from the directory
    documents = loader.load()

    # Initialize a text splitter with specified parameters
    text_splitter = RecursiveCharacterTextSplitter(chunk_size=chunk_size, chunk
    # Split the documents into chunks
    return text_splitter.split_documents(documents)

# Main function to process documents and upload them to a knowledge base
def embed_and_upload(Knowledge_base_name):
    """
    Process documents from a directory and upload them to a knowledge base.

    Args:
        Knowledge_base_name (str): The name of the knowledge base.
    """
    # Define the directory path based on the knowledge base name
    directory = f"./myFiles/{Knowledge_base_name}"
    # Split the documents into chunks
    files = split_docs(directory)

    # Create or retrieve a knowledge base
    my_kb = create_knowledge_box(Knowledge_base_name)
    # Initialize a sentence embedding model
    model_bge = SentenceTransformer("BAAI/bge-base-en")

    # Process each document chunk
    for i, file in enumerate(tqdm(files, desc="Processing files")):
        # Encode the document content into vectors
```

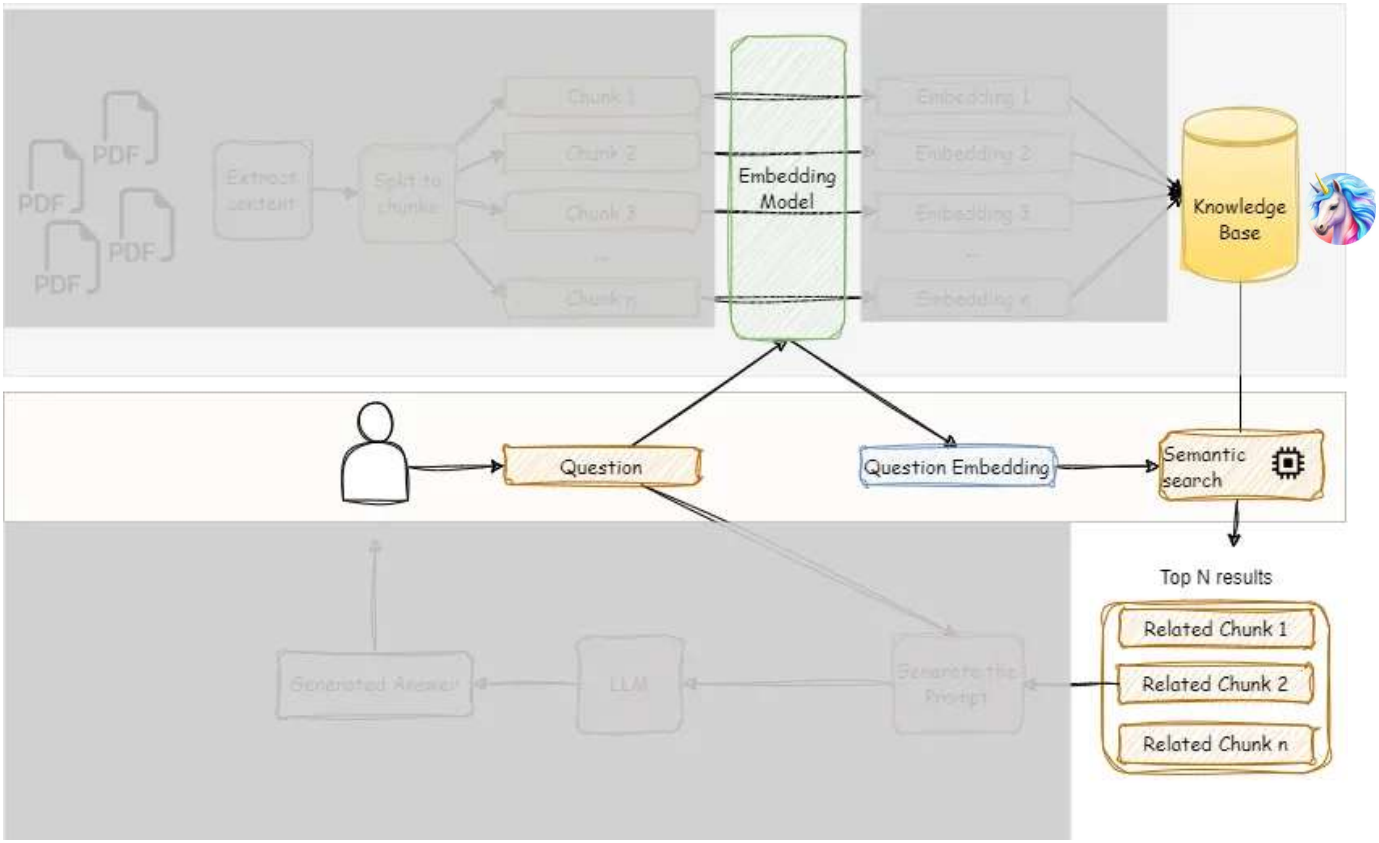
```
            vectors = model_bge.encode([file.page_content])
            # Upload the document to the knowledge base
            my_kb.upload(
                key=f"mykey{i}",
                text=file.page_content,
                labels=[file.metadata["source"]],
                vectors={"bge": vectors[0]},
            )

    if __name__ == "__main__":
        # Process documents for Knowledge Base 1
        Knowledge_base_name_1 = "knowledge_base1"
        embed_and_upload(Knowledge_base_name_1)

        # Process documents for Knowledge Base 2
        Knowledge_base_name_2 = "knowledge_base2"
        embed_and_upload(Knowledge_base_name_2)
```

> *You will need to run this file only once before you can chat with your data.*

Now since we finished the first part of the pipeline, we will be working on handling the user question and generating an answer.

4. User's Questions and Generating an Answer

Alright, now let's dive into the exciting part where we handle user questions and generate an answer from our knowledge base. I will skip the part that talks about Streamlit and focus on the pipeline we have in hand. (Don't worry about Streamlit, you can steal the code and play with it :D)



User Interaction in Streamlit:

Your chatbot interface is built with Streamlit, which provides a convenient way for users to interact. When a user has a question or query, they can input it into the chat interface. This interaction is initiated in the `chat_input` function, which listens for user input:

```
if query := st.chat_input(disabled=not is_api_key_valid):
    st.session_state.messages.append({"role": "user", "content": query})
```

This code snippet captures the user's question and appends it to the chat history.

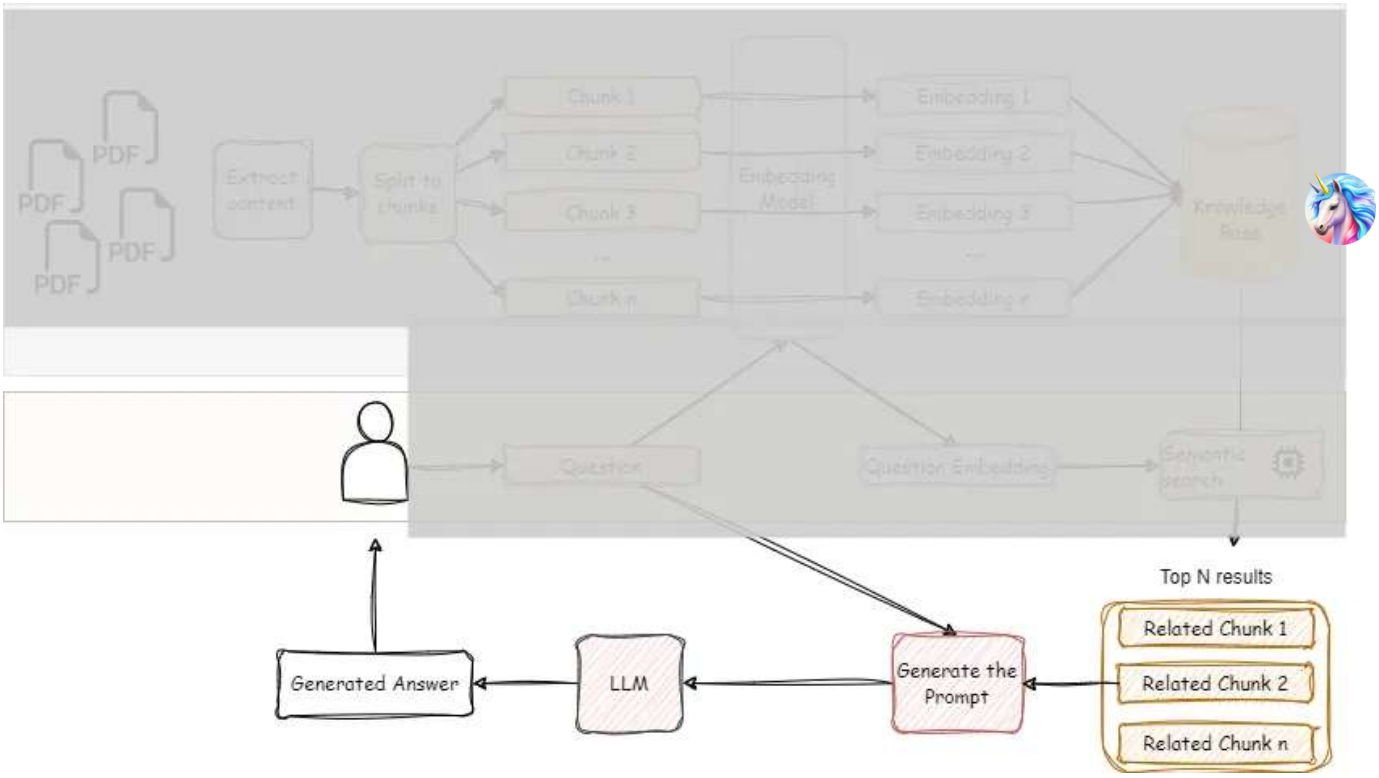
Embedding user's questions and finding related chunks

In this code segment, the user's questions are embedded into numerical vectors using the Sentence Transformers model. These vectors serve as the basis for searching the knowledge base. The code initiates a search with the encoded query, specifying a confidence threshold to filter relevant results. It iterates through the search outcomes, accumulating context that includes text chunks, their sources, and scores. To prevent token-related issues, there's a mechanism in place to ensure the total token count doesn't exceed the model's limits. This context-building process

enriches the chatbot's responses, allowing it to provide informed and context-aware answers to user queries.

```
def find_match(input, turbo_16k, knowledge_base):
    confidence_threshold = 0.85
    reference_number = 4
    query_vectors = model.encode([input])
    my_kb = get_or_create(knowledge_base)
    results = my_kb.search(
        vector=query_vectors[0],
        vectorset="bge",
        min_score=confidence_threshold,
        page_size=reference_number)

    context = {"returned_text": [],
               "source": [],
               "score": []}
    total_tokens = 0
    max_tokens_allowed = 16385 if turbo_16k else 4097
    for result in results:
        if total_tokens < max_tokens_allowed:
            context["returned_text"].append(result.text)
            context["source"].append(result.labels[0])
            context["score"].append(result.score)
            total_tokens += len(result.text)
    return context
```

Generating the Answer



Before we can generate any response, we need to connect to the OpenAI API

OpenAI Model Integration

```
openai.api_key = os.environ['OPEN_AI_KEY']
llm = ChatOpenAI(model_name=model_used, openai_api_key=os.environ['OPEN_AI_KEY'
```

Once the user's question is received, your code proceeds to generate a response. This happens in the `generate_response` function

```
conversation = ConversationChain(memory=st.session_state.buffer_memory, prompt=

response, context = YourDataChat.generate_response(conversation, query, model_u
```

- It first prepares the context and uses it to generate a response. The context includes information about the user's query and the knowledge base.

- It uses the `ConversationChain` to maintain context and generate responses. The `conversation.predict` method is used to predict the chatbot's response.

Run the Code

To run the code from GitHub, create a .env file and add the following:

```
OPEN_AI_KEY = <Your_API_Key>
ROBOT_LOGO = 
USER_LOGO = 
```

Conclusion:

In conclusion, our expedition through the creation of a data-savvy chatbot has been an exhilarating journey into the heart of conversational AI. With a deep dive into the essential tools of Streamlit, OpenAI API, and Vector DB, we've not only demystified the process but also rolled up our sleeves to code our way to success. From managing and processing data files to embedding crucial insights and responding to user queries, we've laid the foundation for an intelligent and user-friendly chatbot. As we wrap up this adventure, let's remember that the world of conversational AI continues to evolve, offering endless opportunities for innovation. Armed with the knowledge gained here, you're now ready to embark on your own data conversation revolution, sparking meaningful dialogues and insights with your chatbot creation. Feel free to explore the code and resources for this project in our GitHub repository: GitHub Repository. Happy coding, and may your chatbot journey be as exciting as the one we've shared! 🤖💬🚀

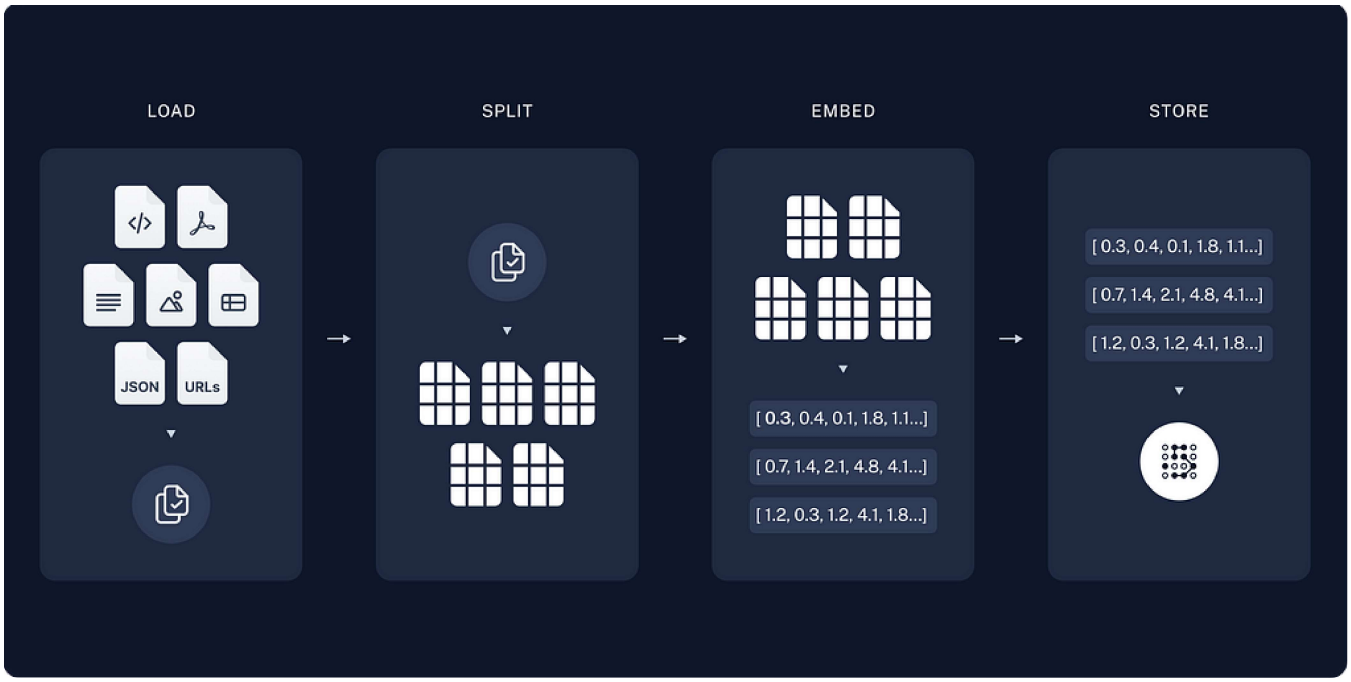
Data Science | LIm | Chatbots | Streamlit | Vector Database

Follow

Written by mutaz younes

3 Followers

Recommended from Medium



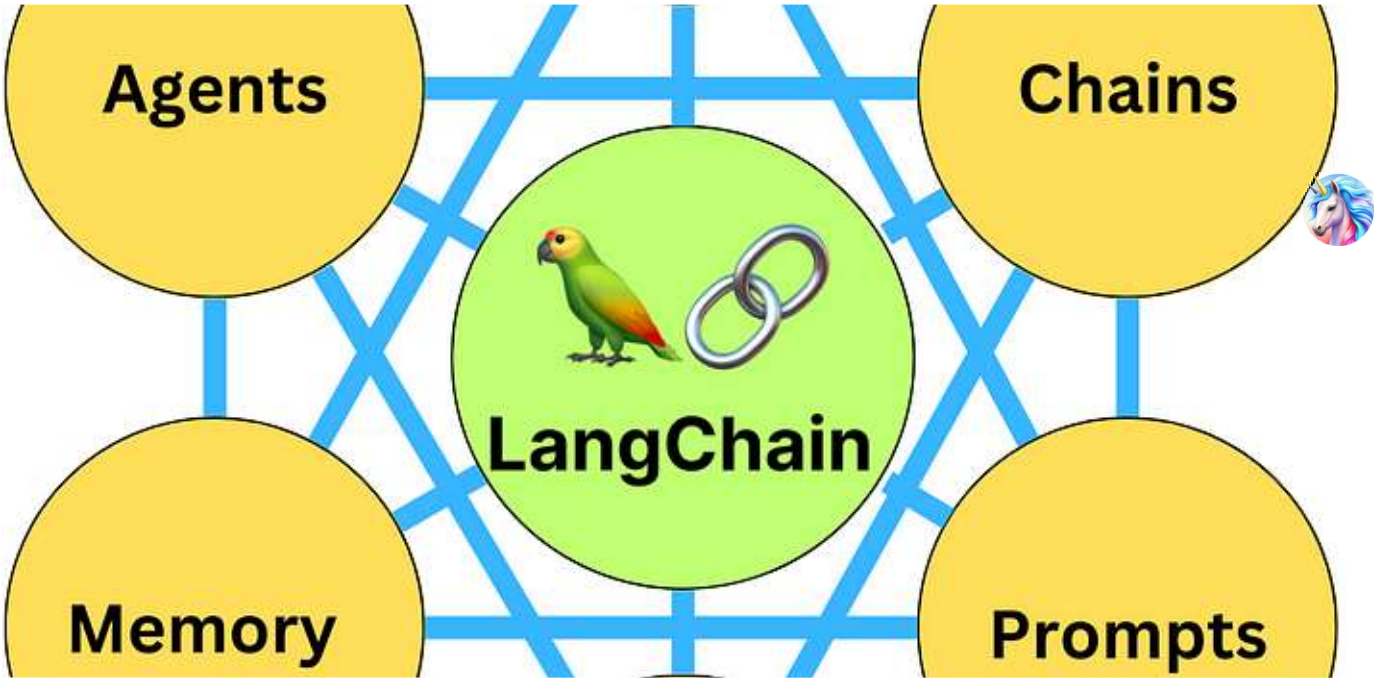
Param Mehta in USF-Data Science

A LangChain chatbot using PDFs

Table of j

14 min read · Feb 1, 2024

363 1



Dash ICT

Build Chatbot with LLMs and LangChain 🦜🔗

In this article I share my experience in building Chatbot through my work at Dash Company, Our goal is to delve into a comprehensive...

11 min read · Aug 20, 2023

762 4

Lists

Predictive Modeling w/ Python

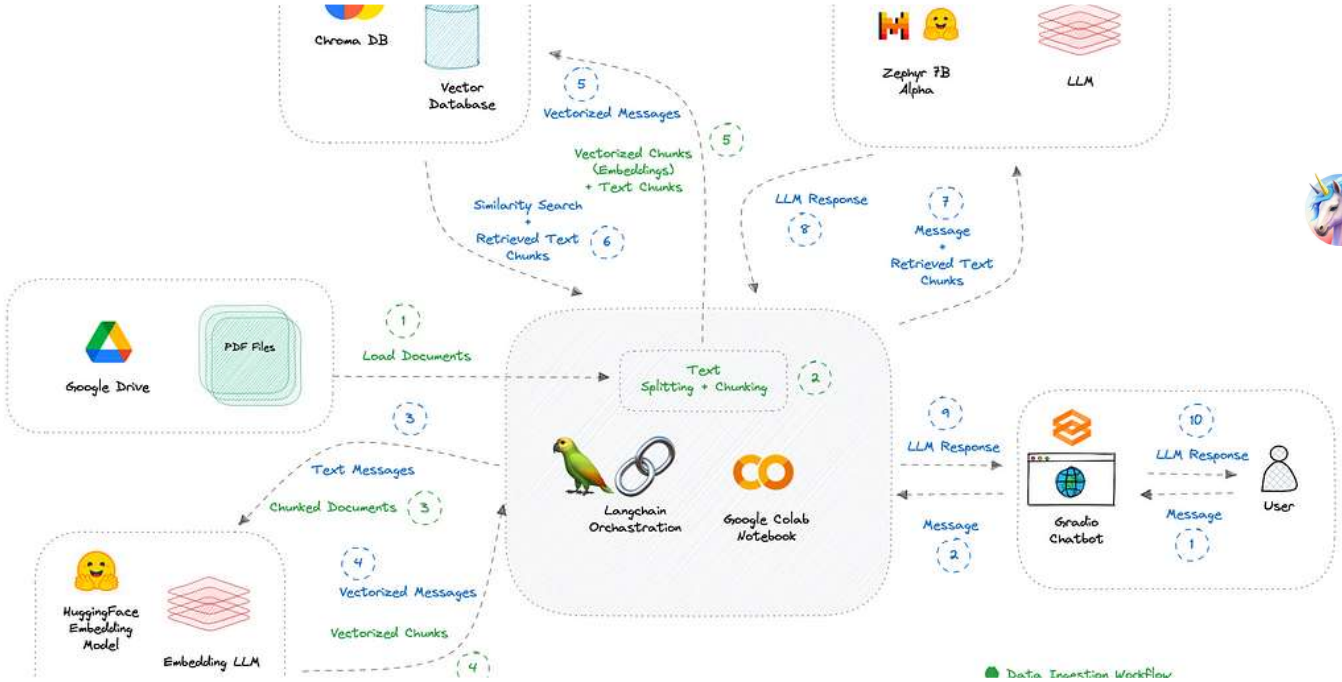
20 stories · 899 saves

Natural Language Processing

1190 stories · 665 saves

Practical Guides to Machine Learning

10 stories · 1050 saves



ai geek (wishesh) in Level Up Coding

Building a Private AI Chatbot

Chat with PDF using Google Colab, Zephyr 7B Alpha, ChromaDB, HuggingFace, and Langchain. It's free and it works like a charm.

5 min read · Oct 22, 2023

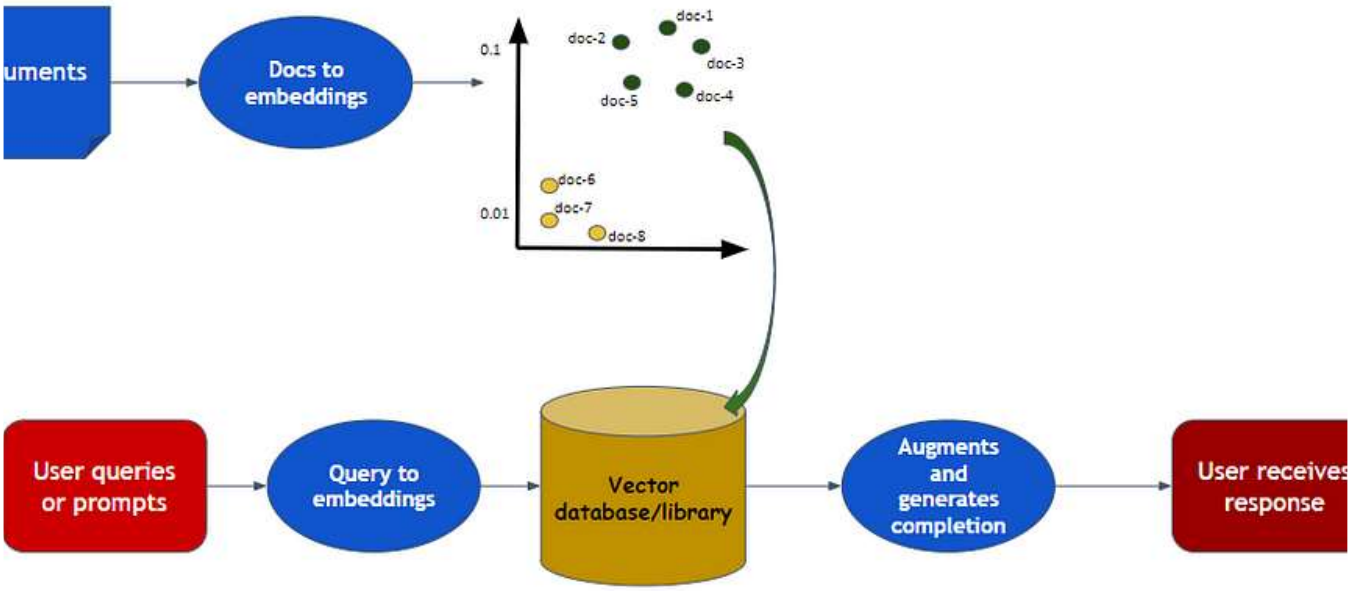
Diptiman Raichaudhuri

Rapid Q&A on multiple PDFs using langchain and chromadb as local disk vector store

Disclosure: All opinions expressed in this article are my own, and represent no one but myself and not those of my current or any previous...

11 min read · Sep 25, 2023

91 3



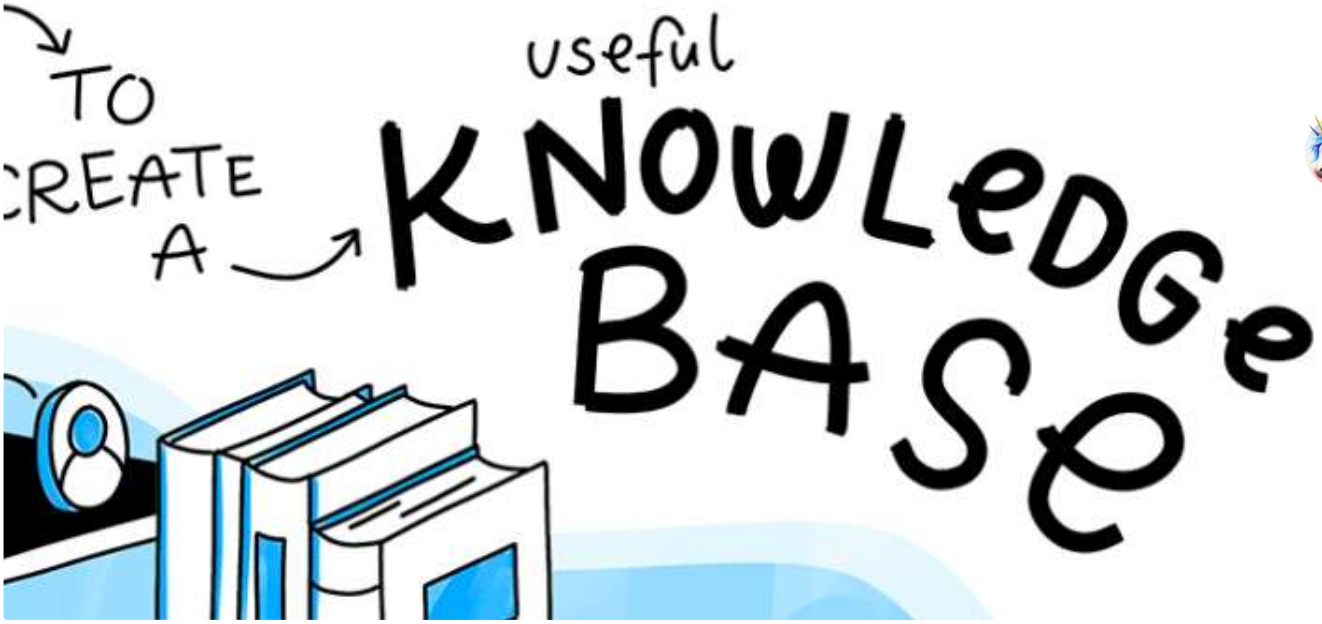
Akriti Upadhyay in Accredian

Implementing RAG with Langchain and Hugging Face

Using Open Source for Information Retrieval

9 min read · Oct 16, 2023

647 11



Isidoro Grisanti

Query your knowledge base easily with OpenSearch + LangChain

An approach on how to query your knowledge base by using natural language, leveraging on AWS OpenSearch & LangChain

5 min read · Oct 13, 2023