Implementing RAG in LangChain with Chroma: A Step-by-Step Guide

Callum Macpherson · Follow

11 min read · Apr 28, 2024

Listen Share More

Disclaimer: I am new to blogging. So, if there are any mistakes, please do let me know. All feedback is warmly appreciated.

Figure 1: AI Generated Image with the prompt “An AI Librarian retrieving relevant information”

Introduction

In natural language processing, Retrieval-Augmented Generation (RAG) has emerged as a powerful technique for generating contextually relevant responses that are dense in information. This technique is often used when building LLM systems that are required to retrieve information from large bodies of text that are rich in complex information. In this blog post, we will explore how to implement RAG in LangChain, a useful framework for simplifying the development process of applications using LLMs, and integrate it with Chroma to create a vector database. This step-by-step guide will walk you through the process of setting up the environment, preparing the data, implementing RAG, and creating a vector database with Chroma.

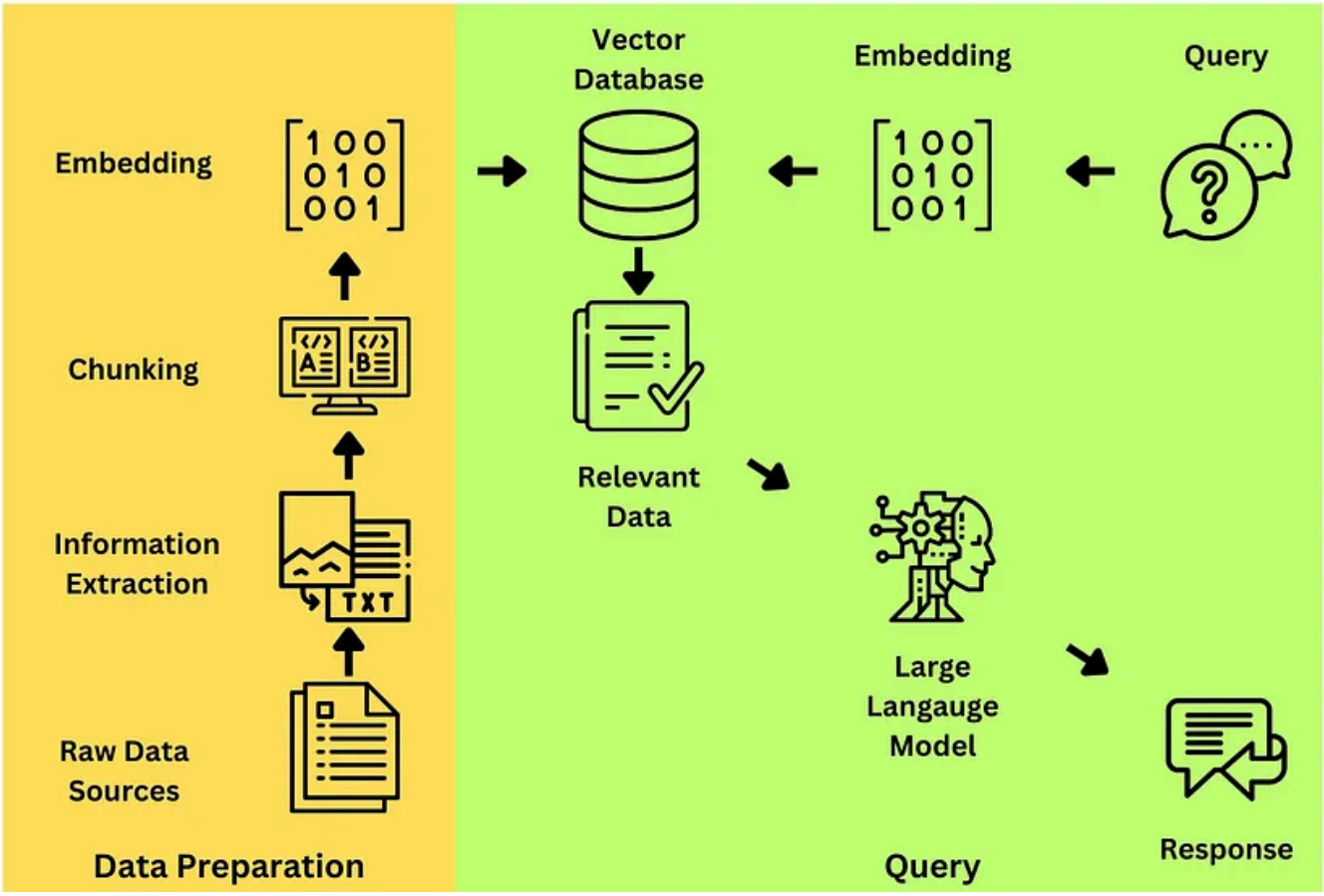


Figure 2: Retrieval Augmented Generation (RAG): overview

Figure 2shows an overview of RAG. The first step is data preparation (highlighted in yellow) in which you must:

1. Collect raw data sources.

2) Extract the raw text data (using OCR, PDF, web crawlers etc.).

3) Split the text into appropriate length chunks.

4) Compute an embedding to be stored in the vector database.

Once the vector database has been created, you can query the system (highlighted in green):

1) A user can query the system with raw text.

2) An embedding is computed for the query.

3) Search the vector database for contextually relevant data.

4) Give the relevant context and query to an LLM

5) Return a contextually enriched response to the user.

Understanding RAG and LangChain

This approach combines retrieval-based methods with generative models to produce responses that are not only coherent but also contextually relevant. LangChain provides a flexible and scalable platform for building and deploying advanced language models, making it an ideal choice for implementing RAG, but another useful framework to use is LlamaIndex.

RAG involves supplementing an LLM with additional information retrieved from elsewhere to improve the model's responses. There are many different approaches to deploying an effective RAG system. Finding the most effective system requires extensive experimentation to optimize each component, including data collection, model embeddings, chunking method and prompting templates. Nonetheless, most systems follow similar design principles and I will provide that overview in the blog post.

Setting up the Environment

Before diving into the implementation, you will need to set up your development environment. Install the relevant dependencies and activate this environment.

Prerequisites:

Before you begin, make sure you have the following installed on your system:

- Python (3.x recommended)
- virtualenv or venv (usually comes pre-installed with Python)

Steps:

- Clone your project repository from the remote repository using Git. If you don't have a repository yet, create one and initialize it with your project files.

```
git clone <repository_url>
cd <project_directory>
```

- Create a Virtual Environment: Navigate to your project directory and create a new virtual environment using either virtualenv or venv. Replace <environment_name> with your desired environment name.

```
# Using venv (Python 3)
python3 -m venv <environment_name>
```

- Activate the Environment: Activate the virtual environment you just created

```
# On Windows
<environment_name>\Scripts\activate
# On Unix or MacOS
source <environment_name>/bin/activate
```

- Install Dependencies: Install project dependencies using pip. You can do this by using the requirements.txt file.

```
# Use the requirements.txt file
pip install -r requirements.txt
```

- Work on the Project: You can run the project within the activated virtual environment. All dependencies will be isolated to this environment, ensuring no conflicts with other projects.

Implementation: Step-by-Step

1: Import Dependencies

```
# Langchain dependencies
from langchain.document_loaders.pdf import PyPDFDirectoryLoader # Importing PDF
from langchain.text_splitter import RecursiveCharacterTextSplitter # Importing
from langchain.embeddings import OpenAIEmbeddings # Importing OpenAI embeddings
from langchain.schema import Document # Importing Document schema from Langchai
from langchain.vectorstores.chroma import Chroma # Importing Chroma vector stor
from dotenv import load_dotenv # Importing dotenv to get API key from .env file
```

```
from langchain.chat_models import ChatOpenAI # Import OpenAI LLM
import os # Importing os module for operating system functionalities
import shutil # Importing shutil module for high-level file operations
```

2: Loading in the Data

For further information on how to load data from other types of files see the LangChain docs. Full guides can be found on loading in files such as `.csv`, `.html`, `.json` and more!

```
# Directory to your pdf files:
DATA_PATH = "/data/"
def load_documents():
  """
  Load PDF documents from the specified directory using PyPDFDirectoryLoader.
  Returns:
  List of Document objects: Loaded PDF documents represented as Langchain
                                                    Document objects.
  """
  # Initialize PDF loader with specified directory
  document_loader = PyPDFDirectoryLoader(DATA_PATH)
  # Load PDF documents and return them as a list of Document objects
  return document_loader.load()

documents = load_documents() # Call the function
# Inspect the contents of the first document as well as metadata
print(documents[0])
```

3: Chunking

In RAG systems, “chunking” refers to the segmentation of input text into shorter and more meaningful units. This enables the system to efficiently pinpoint and retrieve relevant pieces of information. The quality of chunks are **essential** to how effective your system will be.

The most important thing to consider when deciding a chunking strategy is the structure of the documents that you are loading into your vector database. If the documents contain similar-length paragraphs, it would be useful to consider this when determining the size of the chunk.

```
def split_text(documents: list[Document]):
  """
  Split the text content of the given list of Document objects into smaller chu
  Args:
    documents (list[Document]): List of Document objects containing text conter
  Returns:
    list[Document]: List of Document objects representing the split text chunks
  """
  # Initialize text splitter with specified parameters
  text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=300, # Size of each chunk in characters
    chunk_overlap=100, # Overlap between consecutive chunks
    length_function=len, # Function to compute the length of the text
    add_start_index=True, # Flag to add start index to each chunk
  )

  # Split documents into smaller chunks using text splitter
  chunks = text_splitter.split_documents(documents)
  print(f"Split {len(documents)} documents into {len(chunks)} chunks.")

  # Print example of page content and metadata for a chunk
  document = chunks[0]
  print(document.page_content)
  print(document.metadata)

  return chunks # Return the list of split text chunks
```

If your `chunk_size` is too large, then you will be inputing a lot of noisy and unwanted context to the final LLM query. Further, as LLMs are limited by their context window size, the larger the chunk, the fewer pieces of relevant information you can include as context to your query.

The `chunk_overlap` refers to intentionally duplicating tokens at the beginning and end of each adjacent chunk. This helps retain additional context from neighbouring chunks which can improve the quality of the information passed into the prompt.

When deploying these systems to real-world applications, it is important to plot distributions of text lengths in your data, and tune these parameters based on experimentation of parameters such as `chunk_size` and `chunk_overlap`.

4: Creating a Vector Database with Chroma

In order to turn your text data into embeddings, you need an *embedding model*. This is a pre-trained model to transform the text into a vector of numbers. This is a high-dimensional representation of the text in embedding space, where each dimension corresponds to a specifific semantic or syntactic feature of the word.

The choice of the embedding model used impacts the overall efficacy of the system, however, some engineers note that the choice of embedding model often has less of an impact than the choice of chunking strategy. Further, selecting the largest and most powerful LLMs is not always the most effective for creating the vector database. The same engineers found that smaller models can outperform top-ranked state-of-the-art models for specific tasks. Therefore, you may be able to reduce computational cost without sacrificing on overall system effectiveness.

In the example provided, I am using Chroma because it was designed for this use case. However, there are some tools to use relational databases such as PostgreSQL. We have used OpenAIEmbeddings API which requires an API key:

```
# Path to the directory to save Chroma database
CHROMA_PATH = "chroma"
def save_to_chroma(chunks: list[Document]):
  """
  Save the given list of Document objects to a Chroma database.
  Args:
  chunks (list[Document]): List of Document objects representing text chunks tc
  Returns:
  None
  """

  # Clear out the existing database directory if it exists
  if os.path.exists(CHROMA_PATH):
    shutil.rmtree(CHROMA_PATH)

  # Create a new Chroma database from the documents using OpenAI embeddings
  db = Chroma.from_documents(
    chunks,
    OpenAIEmbeddings(),
    persist_directory=CHROMA_PATH
  )

  # Persist the database to disk
  db.persist()
  print(f"Saved {len(chunks)} chunks to {CHROMA_PATH}.")
```

I encourage experimentation with open-sourced models such as Llama3 (try the 8B parameter version first, especially if your system needs to be 100% local). If you are working on a very niche or nuanced use case, off-the-shelf embedding models may not be useful. Therefore, you might want to investigate fine-tuning the embedding model on the domain data to improve the retrieval quality.

```
def generate_data_store():
  """
  Function to generate vector database in chroma from documents.
  """
  documents = load_documents() # Load documents from a source
  chunks = split_text(documents) # Split documents into manageable chunks
  save_to_chroma(chunks) # Save the processed data to a data store

# Load environment variables from a .env file
load_dotenv()
# Generate the data store
generate_data_store()
```

5: Query the vector database and the LLM

First, we need to identify what question we need the answer from our PDF. The PDF used in this example was my MSc Thesis on using Computer Vision to automatically track hand movements to diagnose Parkinson's Disease. Therefore, let's ask the system to explain one of the methods used in the paper. I used an object detection method to predict bounding box coordinates around the hand in the image to remove unnecessary pixels, the method used was YOLOv4 (I still can't believe we are on YOLOv8 already😆). Therefore, we will ask the model to explain how the YOLO method works:

```
query_text = "Explain how the YOLO method works"
```

Ok, now that we have defined our query, we need to define a prompt template to input to the LLM that will generate the final response. As you can see by the `PROMPT_TEMPLATE` variable below, the structure is intuitive and simple. First, we provide `{context}` to the LLM which is the information that we have **retrieved** from our chroma database. Then, we **augment** the context with our `{question}` to

the LLM to **generate** a contextually enhanced response. It's way simpler than it sounds, right?

```
PROMPT_TEMPLATE = """
Answer the question based only on the following context:
{context}
 - -
Answer the question based on the above context: {question}
"""
```

We have defined our query and prompt template. Let's put it all together!

In RAG systems, finding the most useful and relevant information in the database as fast as possible are the two objectives you want to optimize for when finetuning. The main approaches for retrieving data from the vector database are:

1) **Term-based matching**: identifies keywords from questions and matches them to relevant chunks using term statistics. This method is usually simpler and requires less preprocessing.

2) **Vector-similarity search**: computes a distance metric between the query vectors and indexed vectors in the database. This method is more effective typically for retrieving contextually relevant information to the prompt.

3) **Hybrid search**: integrates term-based and vector similarity for more comprehensive results.

In this example, we are going to use Vector-similarity search. However, it is strongly advised that the optimal method and parameters are found experimentally to tailor the system to your domain and use case. Some other tuneable parameters to be found experimentally are:

- the relevance score threshold (we set ours to 0.7 arbitrarily)
- the LLM (base vs. tuned)
- prompt template

```
def query_rag(query_text):
  """
  Query a Retrieval-Augmented Generation (RAG) system using Chroma database anc
  Args:
    - query_text (str): The text to query the RAG system with.
  Returns:
    - formatted_response (str): Formatted response including the generated text
    - response_text (str): The generated response text.
  """
  # YOU MUST - Use same embedding function as before
  embedding_function = OpenAIEmbeddings()

  # Prepare the database
  db = Chroma(persist_directory=CHROMA_PATH, embedding_function=embedding_funct

  # Retrieving the context from the DB using similarity search
  results = db.similarity_search_with_relevance_scores(query_text, k=3)

  # Check if there are any matching results or if the relevance score is too lc
  if len(results) == 0 or results[0][1] < 0.7:
    print(f"Unable to find matching results.")

  # Combine context from matching documents
  context_text = "\n\n - -\n\n".join([doc.page_content for doc, _score in resul

  # Create prompt template using context and query text
  prompt_template = ChatPromptTemplate.from_template(PROMPT_TEMPLATE)
  prompt = prompt_template.format(context=context_text, question=query_text)

  # Initialize OpenAI chat model
  model = ChatOpenAI()

  # Generate response text based on the prompt
  response_text = model.predict(prompt)

   # Get sources of the matching documents
  sources = [doc.metadata.get("source", None) for doc, _score in results]

  # Format and return response including generated text and sources
  formatted_response = f"Response: {response_text}\nSources: {sources}"
  return formatted_response, response_text

# Let's call our function we have defined
formatted_response, response_text = query_rag(query_text)
# and finally, inspect our final response!
print(response_text)
```

```
'The YOLO method works by splitting the image into a grid and using a regression-based approach to predict bounding
boxes for objects within each grid cell. Each cell can only predict up to two objects from one class, which can pose
spatial constraints and make it difficult to detect multiple small objects in close proximity. This design choice
can make it challenging for the model to accurately detect entire groups of objects, such as a flock of birds, in
the image space.'
```

Figure 3: RAG System output to query: “Explain how the YOLO method works”

The generated response accurately describes YOLO using information retrieved from my thesis! The prototype has been built and seems to work well. There are several questions that we are now left with as engineers:

- How well does the system work?
- How would we even go about quantifying how much better the RAG-enabled LLM was vs. the original LLM?
- When fine-tuning the RAG parameters (such as chunking method, prompt template, LLM agent used) how do we know if the changes made have improved or worsened the responses?

6: Evaluation

We have created a prototype using arbitrary values. The next project we would need to undertake would be finding the optimal combination of chunking logic, embedding model, LLM response agent. Generative tasks such as this are extremely difficult to quantitatively evaluate. Therefore, we need to propose a method for this to be implemented in a future blog.

We have a complex and multi-stage system with many interchangeable parts. Therefore, it is recommended that we undertake an evaluation of each component in isolation and end-to-end evaluation:

- **Component-wise evaluation**: for example compare embedding methods, retrieval methods, LLM response methods, and even the raw data sources.

- **End-to-end evaluation**: assess the quality of the entire system.

Further, I recommend writing unit tests that query an LLM with expected and actual responses from the RAG system. For example, we could use the following evaluation prompt, logic and unit test:

```
# Template for evaluation prompt
EVAL_PROMPT = """
Expected Response: {}
Actual Response: {}
###
(Answer with 'True' or 'False: Does the actual response match the expected resp
"""

# Strip leading/trailing whitespaces and convert to lowercase
retrieval_test_response = evaluation_results_str.strip().lower()

# Check if "true" or "false" is present in the response
if "true" in retrieval_test_response:
  return True
elif "false" in retrieval_test_response:
  return False
else:
  # Raise an error if neither "true" nor "false" is found
  raise ValueError("Did not return true or false")

  def test_retrieval_response():
  # Test function to validate retrieval response
  assert query_and_validate(
    prompt="a question about our data",
    expected_response="a known fact in our data"
  )
```

There is a chance that a generous LLM might pass incorrect responses, therefore also consider negative test cases. I.e. assert that the expected response (one that is wrong) does not equal the actual response.

Conclusion

In this blog, I have introduced the concept of Retrieval-Augmented Generation and provided an example of how to query a .pdf file using LangChain in Python. I have also introduced the concept of how RAG systems could be finetuned and quantitatively evaluate the responses using unit tests.

Links

Github repo for this blog.

Original [RAG paper].

Connect with me on LinkedIn for enquiries.

The original blog on my website.

Thanks for reading 😎

Llm | Llm Evaluation | Retrieval Augmented Gen | Langchain | Agents

Follow

Written by Callum Macpherson

27 Followers

ML Practitioner who aims to learn publicly

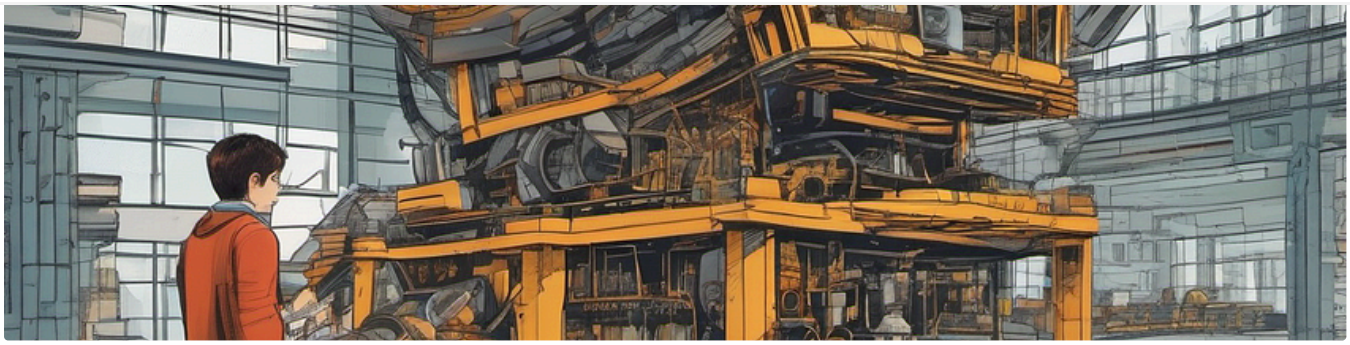
More from Callum Macpherson

Callum Macpherson

Malicious LLM Prompt Detection in Python

I have uploaded the complete code (Python and Jupyter notebook) on GitHub: https://github.com/CallumJMac/prompt_classification

Apr 21 1 1



Callum Macpherson

LLMs Evals: A General Framework for Custom Evaluations

Introduction

May 7

Callum Macpherson

Advanced Retrieval for Retrieval-Augmented Generation

Please see the link to the GitHub Repo at the end of the post!

May 22

Callum Macpherson

Continual Learning: Which metrics are important?

One of the most difficult things to identify throughout the lifecycle of a deployed machine learning model is: “When do I need to retrain...

Apr 17 1

See all from Callum Macpherson

Recommended from Medium

Ming

Comparing LangChain and LlamaIndex with 4 tasks

LangChain v.s. LlamaIndex — How do they compare? Show me the code!

Jan 11 1.2K 9

Rajesh Mani Kumar G

Llama 3.1 Simple RAG using Embedchain via Local Ollama

Llama 3.1- new 128K context length—open source model from Meta with state-of-the-art capabilities in general knowledge, steerability...

Jul 29 48

Lists

Natural Language Processing

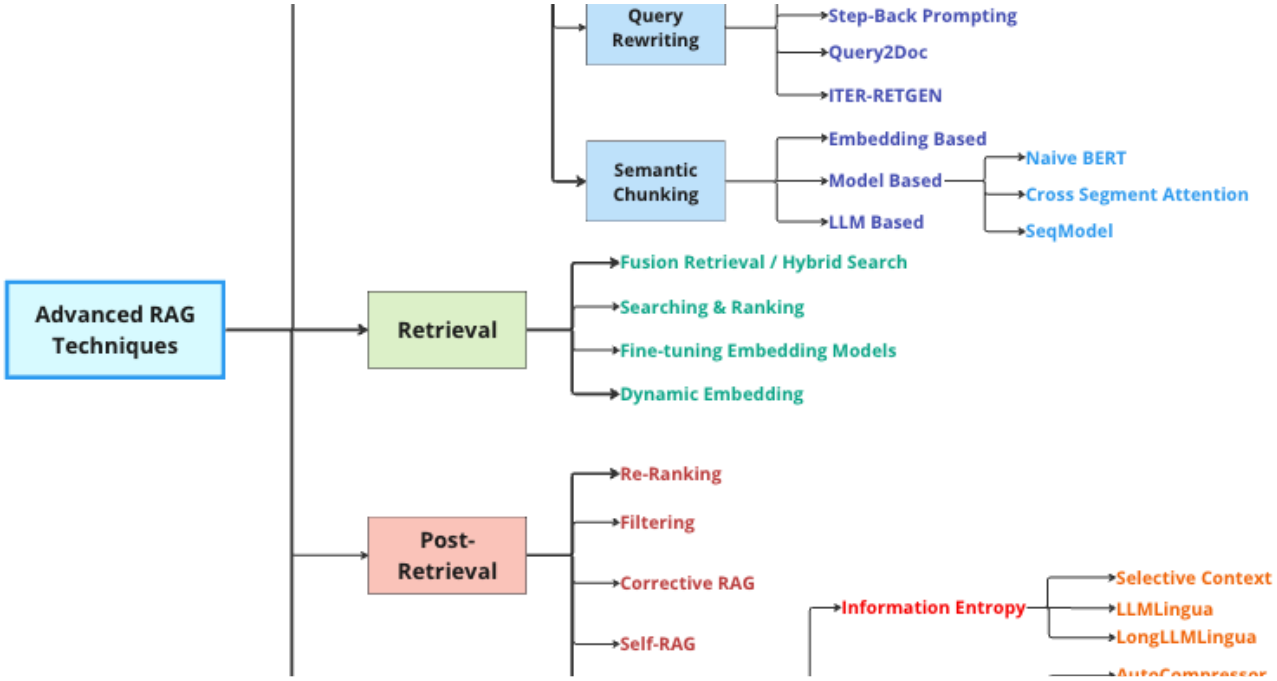
1662 stories · 1239 saves

ChatGPT prompts

48 stories · 1930 saves

Staff Picks

718 stories · 1248 saves

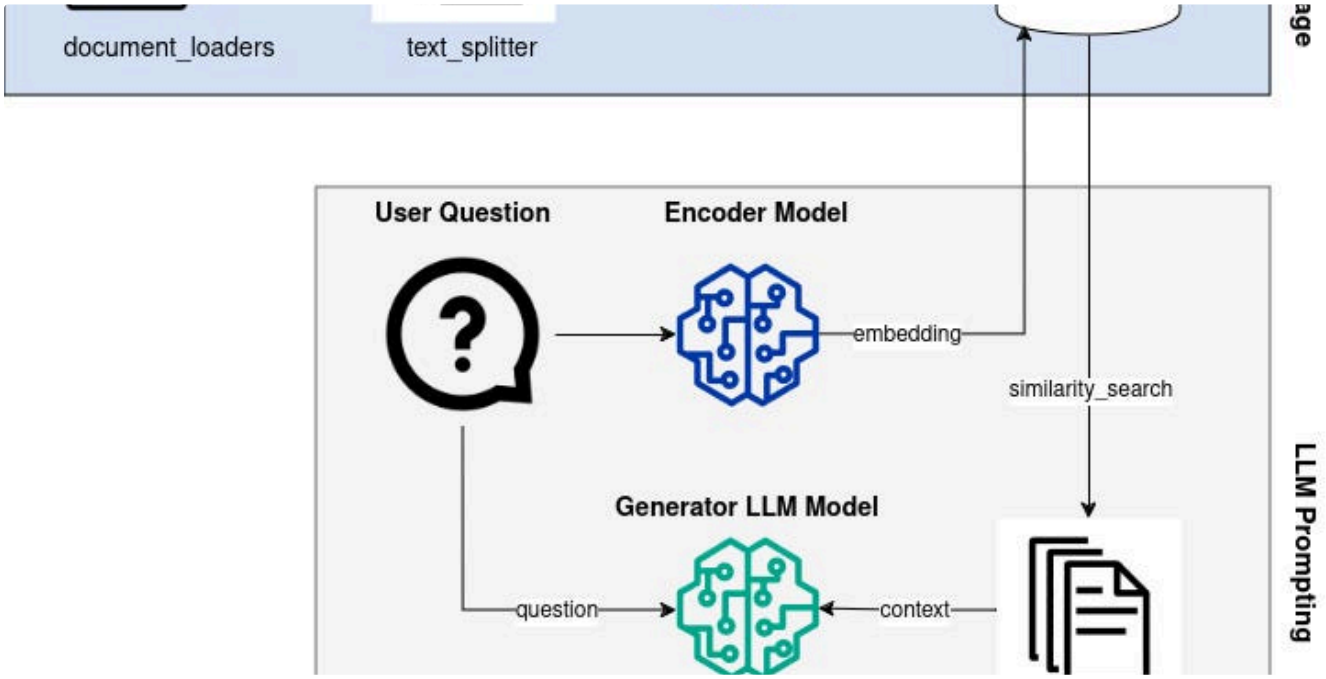


Vipra Singh

Building LLM Applications: Advanced RAG (Part 10)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

Apr 28 659 5

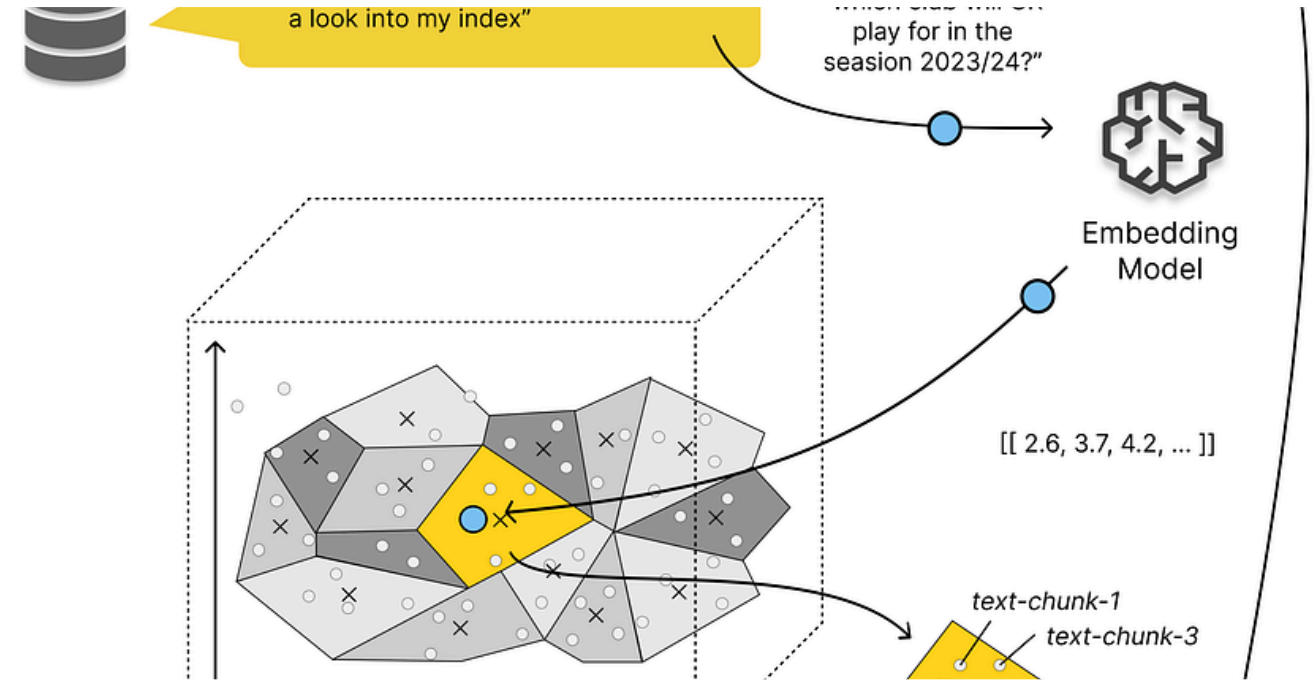


Dr. Leon Eversberg in Towards Data Science

How to Build a Local Open-Source LLM Chatbot With RAG

Talking to PDF documents with Google's Gemma-2b-it, LangChain, and Streamlit

Apr 1 1.6K 11



Dominik Polzer in Towards Data Science

All You Need to Know about Vector Databases and How to Use Them to Augment Your LLM Apps

A Step-by-Step Guide to Discover and Harness the Power of Vector Databases

Sep 18, 2023 1.7K 12

Priyesh Dave

Leveraging Chroma DB for Efficient Text Embedding and Similarity Search

Introduction

Jun 8 3