

[Open in app](#)

Find a home for your writing: [Join Medium's Pub Crawl, March 19](#)



★ Member-only story

Building RAG Application using Gemma 7B LLM & Upstash Vector Database



Youssef Hosni · [Follow](#)

Published in Towards AI

9 min read · Mar 8, 2024



Listen



Share



More

Retrieval-Augmented Generation (RAG) is the concept of providing large language models (LLMs) with additional information from an external knowledge source. This allows them to generate more accurate and contextual answers while reducing hallucinations. In this article, we will provide a step-by-step guide to building a complete RAG application using the latest open-source LLM by Google **Gemma 7B** and **Upstash** serverless vector database.

**Table of Contents:**

1. Getting Started & Setting Up Working Environment
2. Download & Split the Cosmopedia Dataset
3. Generating Embedding with Sentence Transformers Model
4. Store the Embeddings in the Upstash Vector Database
5. Introduce & Use Gemma 7B LLM
6. Querying the RAG Application

You can try Upstash Vector Database for free from here:

Upstash: Serverless Data for Redis and Kafka

Designed for the serverless with per-request pricing and Redis API on durable storage.

```
console.upstash.com
```

1. Getting Started & Setting Up Working Environment

The first step in building an RAG application is to prepare the working environment. We will start with downloading the packages we will use in building the application:

```
%pip install -q -U langchain torch transformers sentence-transformers datasets
```

Next, we will import the packages and libraries that will be used:

```
import torch
from upstash_vector import Vector
from datasets import load_dataset
from tqdm import tqdm, trange
from langchain_community.document_loaders.csv_loader import CSVLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from transformers import AutoTokenizer, AutoModelForCausalLM
from transformers import AutoTokenizer, pipeline
from langchain import HuggingFacePipeline
from langchain.chains import RetrievalQA
```

Retrieval-Augmented Generation (RAG) is the concept to provide LLMs with additional information from an external knowledge source. This allows them to generate more accurate and contextual answers while reducing hallucinations. RAG has two main components:

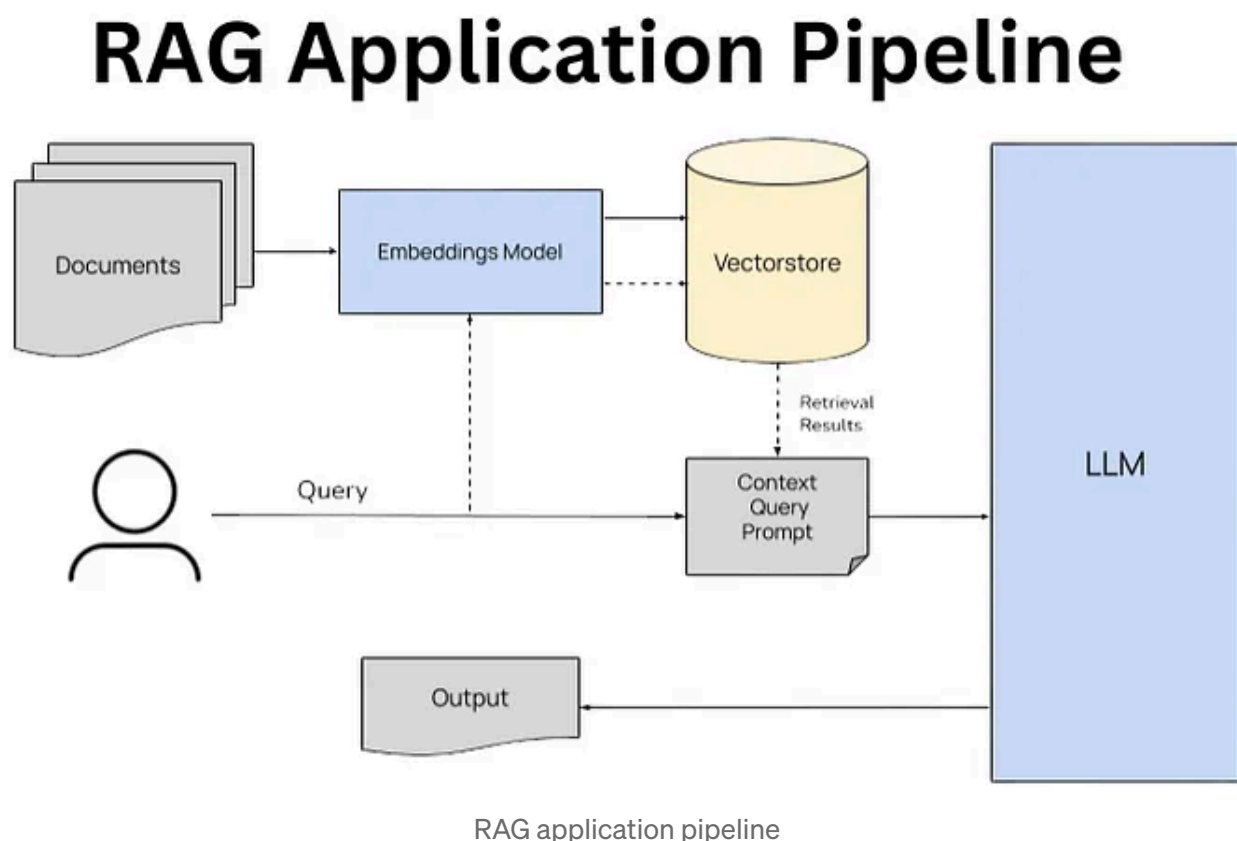
1. **Retrieval:** The IR component acts as a search engine, sifting through a vast amount of text data (typically documents or passages) to identify the most relevant information about a given user query. This process involves techniques like dense passage retrieval, where the system retrieves not just documents but also specific text passages that align with the query's intent.

2. Generation: The retrieved passages are fed into the LLM, which acts as a powerful language-processing engine. The LLM analyzes these passages and leverages its understanding of language to generate a response that addresses the user's query comprehensively.

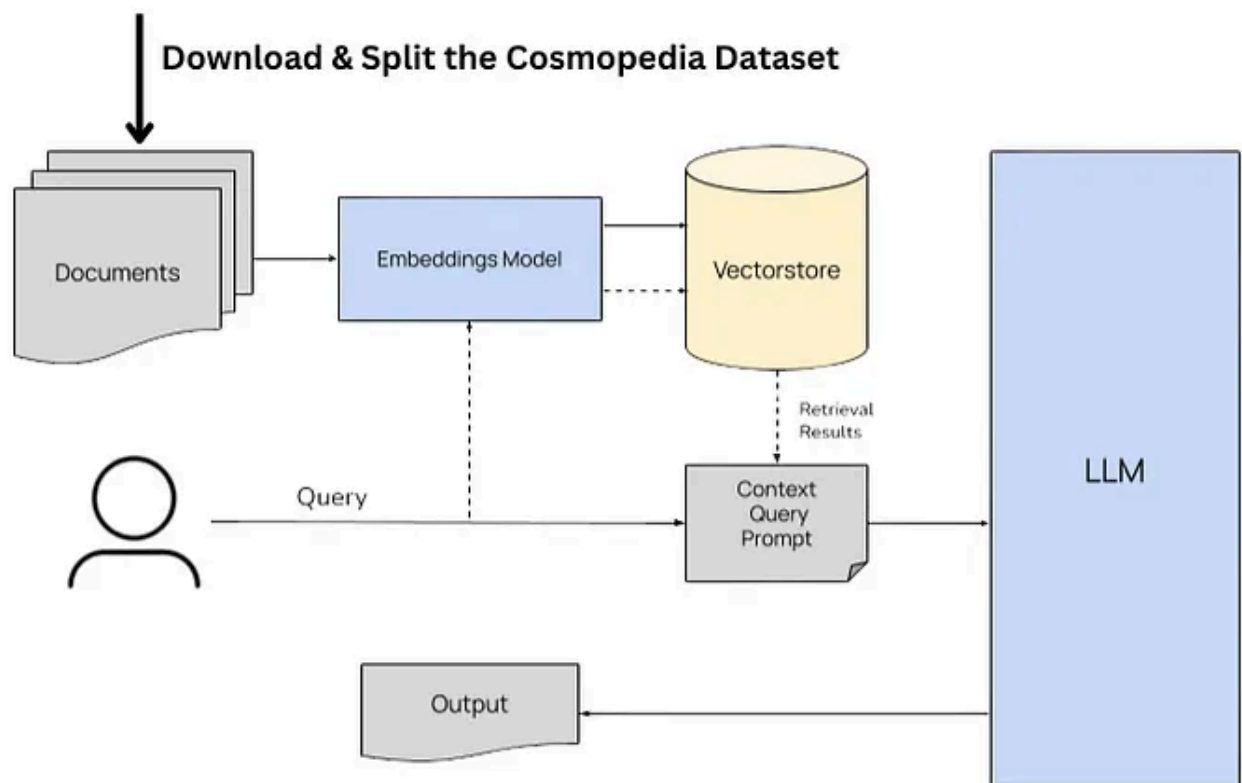
When a user asks a question to the LLM. Instead of asking the LLM directly, we generate embeddings for this query and then retrieve the relevant data from our knowledge library that is well maintained and then use that context to return the answer.

We use vector embeddings (numerical representations) to retrieve the requested document. Once the needed information is found from the vector databases, the result is returned to the user.

This largely reduces the possibility of hallucinations and updates the model without retraining the model, which is a costly process. Here's a very simple diagram that shows the process.



2. Download & Split the Cosmopedia Dataset



Step 1: Download & Split the Cosmopedia Dataset

To build a cutting-edge RAG (Retrieval-Augmented Generation) application, we have strategically chosen the **Cosmopedia dataset** hosted by Hugging Face.

This meticulously compiled dataset comprises a diverse array of synthetic textual sources, ranging from textbooks and blog posts to narratives, social media entries, and WikiHow articles. Generated by the Mixtral-8x7B-Instruct-v0.1 model, Cosmopedia is a monumental collection, boasting over 30 million individual files and an astonishing 25 billion tokens.

Its sheer scale and comprehensiveness mark it as the largest open synthetic dataset currently available. It offers a wealth of material to fuel our innovative endeavors in natural language processing and generation.

The **Cosmopedia dataset** contains 8 subsets:

- **auto_math_text**: 1.95M rows
- **khanacademy**: 24.1k rows
- **Openstax**: 126k rows

- **Stanford:** 1.02M rows
- **Stories:** 4M rows
- **web_samples_v1:** 12.4M
- **web_samples_v2:** 10.3M rows
- **wikiHow:** 179K rows

We will continue working with the ‘Stanford’ subset. We’ll load the dataset using the datasets library.

```
data = load_dataset("HuggingFaceTB/cosmopedia", "stanford", split="train")
```

Next, we will convert it to a Pandas dataframe, and save it to a CSV file.

```
data = data.to_pandas()
data.to_csv("stanford_dataset.csv")
data.head()
```

	text_token_length	prompt	text	seed_data	format	audience
0	527	Write a long and very detailed course unit for...	1.1 Overview of the Course\n\nIn this compreh...	stanford	textbook_narrative	researchers
1	681	Write a long and very detailed course unit for...	Congratulations, fourth-grade friends! You've...	stanford	textbook_narrative	young_children
2	266	Write a long and very detailed course unit for...	9.3 Impact of Ethnic Conflict on Chinese Soci...	stanford	textbook_narrative	college_students
3	1103	Write a long and very detailed course unit for...	1.2 Importance of Premodern Japanese Literatu...	stanford	textbook_narrative	high_school_studnets
4	894	Write a long and very detailed course unit for...	5.1 Recap of the Course: Putting It All Toget...	stanford	textbook_narrative	high_school_studnets

After saving the dataset on our system, we will use the LangChain CSVLoader method to load this dataset.

```
loader = CSVLoader(file_path='./stanford_dataset.csv')
data = loader.load()
```

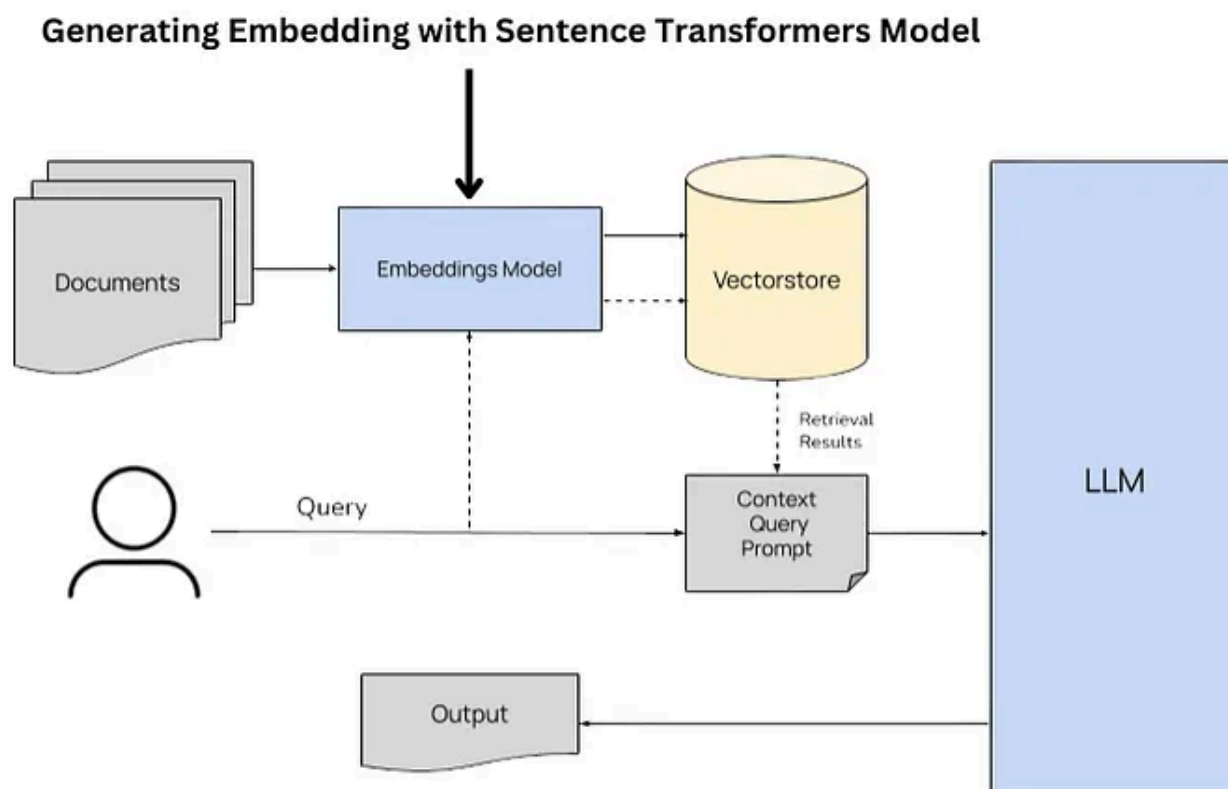
Now that the data is loaded, we need to split the documents inside the data into smaller chunks that can fit into your model's context window.

When you want to deal with long pieces of text, it is necessary to split them into chunks. As simple as this sounds, there is a lot of potential complexity here. Keep the semantically related pieces of text together.

LangChain has many built-in document transformers, making it easy to split, combine, filter, and otherwise manipulate documents. We will use the **RecursiveCharacterTextSplitter** which recursively tries to split by different characters to find one that works with. We will set the **chunk size = 1000** and **chunk overlap = 150**.

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000, chunk_overlap=150)
docs = text_splitter.split_documents(data)
```

3. Generating Embedding with Sentence Transformers Model



Step 2: Generating Embedding with Sentence Transformers Model

The next step is to generate embeddings for the loaded and split text. Generating embeddings is an essential step for building an RAG application as they encode semantic information about text, facilitating understanding of meaning and context crucial for both retrieval and generation processes which enhance retrieval accuracy by comparing query and document embeddings.

We will use the **all-MiniLM-L6-v2** embedding model from the **SentenceTransformers** to generate the embeddings. First, we will initialize it using the code below:

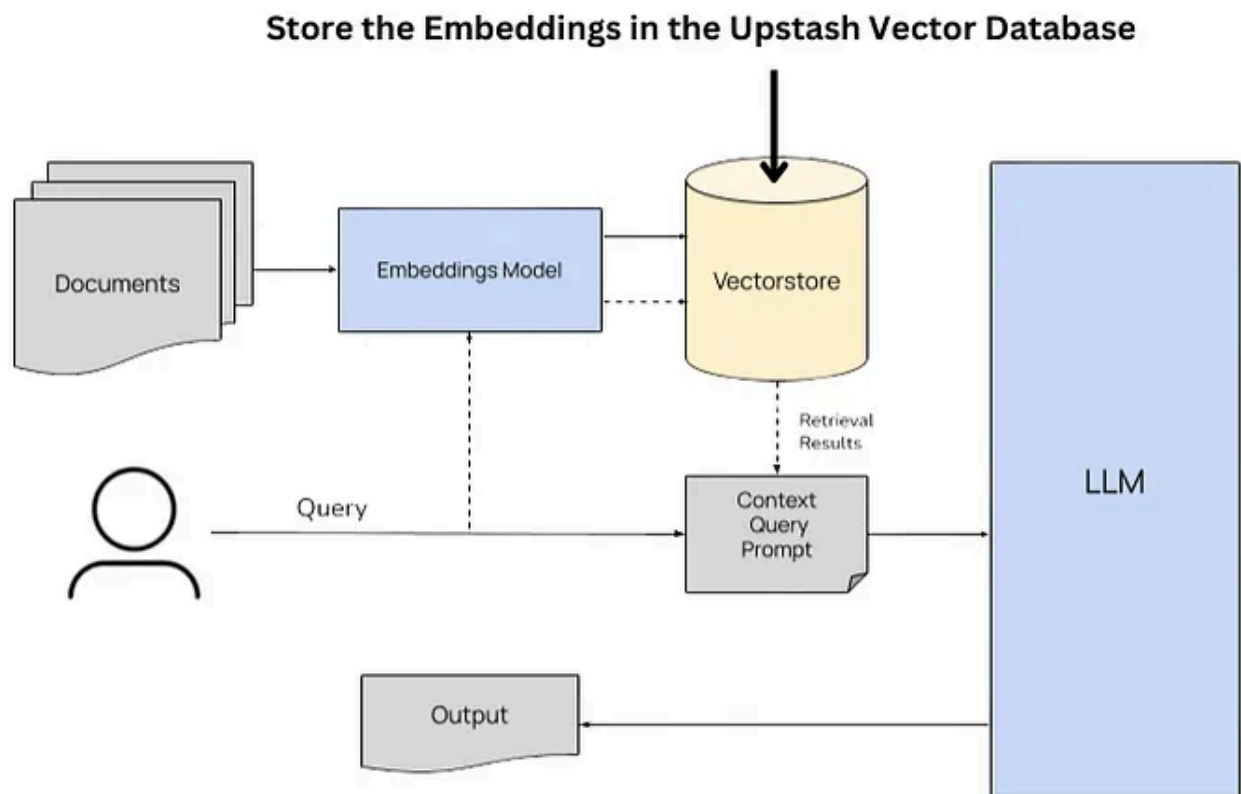
```
modelPath = "sentence-transformers/all-MiniLM-l6-v2"
model_kwargs = {'device':'cpu'}
encode_kwargs = {'normalize_embeddings': False}
embeddings = HuggingFaceEmbeddings(
    model_name=modelPath,
    model_kwargs=model_kwargs,
    encode_kwargs=encode_kwargs
)
```

After initializing it we can now apply it to our documents to generate the embeddings:

```
# Generate embeddings for each chunk of text
chunk_embeddings = []
for doc in docs:
    # Generate embeddings for each chunk
    chunk_embedding = embeddings.encode(doc)
    chunk_embeddings.append(chunk_embedding)
```

Once we generate the embeddings we will need them stored in a vector database. Vector databases support indexing and clustering techniques optimized for high-dimensional data like embeddings, which can further improve retrieval efficiency and accuracy. We will use the **Upstash vector database** to save our embeddings.

4. Store the Embeddings in the Upstash Vector Database



Step 4: Store the Embeddings in the Upstash Vector Database

We will use the **Upstash Vector** database to store the vector embeddings generated before. **Upstash Vector** is a serverless vector database designed for working with vector embeddings.

Since it operates on a serverless model, it abstracts away hosting and management complexities, letting you focus on building LLM applications without further complexities. You will be billed based on API calls.

We will create a free vector database from the **Upstash Console** with **384** dimensions and **Dot_Product** distance. The dimension size is important as it must match the dimensions of the embedding model.

Create IndexSelect a Plan

Name

RAGGemma7BLLM

Name can only contain alphanumeric, underscore, hyphen and dot.

Region

N. Virginia, USA (us-east-1)

For best performance, select the region that is closer to your application.

Dimensions

384

Distance Metric

DOT_PRODUCT

Cancel

Next

After creating the vector index you will get access to the **vector rest URL** and the **vector rest token**. We will define them in the code to be able to access the created vector database.

```
UPSTASH_VECTOR_REST_URL="<YOUR_UPSTASH_VECTOR_REST_URL>"  
UPSTASH_VECTOR_REST_TOKEN="<YOUR_UPSTASH_VECTOR_REST_TOKEN>"
```

To be able to insert the generated embeddings into the vector database, we will have to convert them to **Vector objects** to be inserted into our index. We will use the code below to do so:

```
from tqdm import tqdm, trange
from upstash_vector import Vector

vectors = []

# generate the vectors in batches of 10
batch_count = 10

for i in trange(0, len(chunks), batch_count):
    batch = chunks[i:i+batch_count]

    embeddings = chunk_embedding[batch]

    for i, chunk in enumerate(batch):
        vec = Vector(id=f"chunk-{i}", vector=embeddings[i], metadata={
            "text": chunk
        })

        vectors.append(vec)
```

Once we have generated the vectors we can then upsert all of the vectors to the index at once. Upstash supports for 1000 vectors per request for free indexes

```
from upstash_vector import Index

index = Index(
    url=UPSTASH_VECTOR_REST_URL,
    token=UPSTASH_VECTOR_REST_TOKEN
)

# If you want to reset your index beforehand uncomment this
# index.reset()

index.upsert(vectors)
```

5. Introduce & Use Gemma 7B LLM



Gemma is a family of 4 new LLM models by Google based on Gemini. It comes in two sizes: 2B and 7B parameters, each with base (pretrained) and instruction-tuned versions.

All the variants can be run on various types of consumer hardware, even without quantization, and have a context length of 8K tokens:

- **[gemma-7b](#)**: Base 7B model.
- **[gemma-7b-it](#)**: Instruction fine-tuned version of the base 7B model.
- **[gemma-2b](#)**: Base 2B model.
- **[gemma-2b-it](#)**: Instruction fine-tuned version of the base 2B model.

To use the **[Gemma](#) model**, you should accept the terms on Hugging Face. After that, you must pass the Hugging Face token while logging in.

```
from huggingface_hub import notebook_login
notebook_login()
```

Next, we will initialize the tokenizer and the model

```
model = AutoModelForCausalLM.from_pretrained("google/gemma-7b")
tokenizer = AutoTokenizer.from_pretrained("google/gemma-7b", padding=True, truncation=True)
```

Create a text generation pipeline.

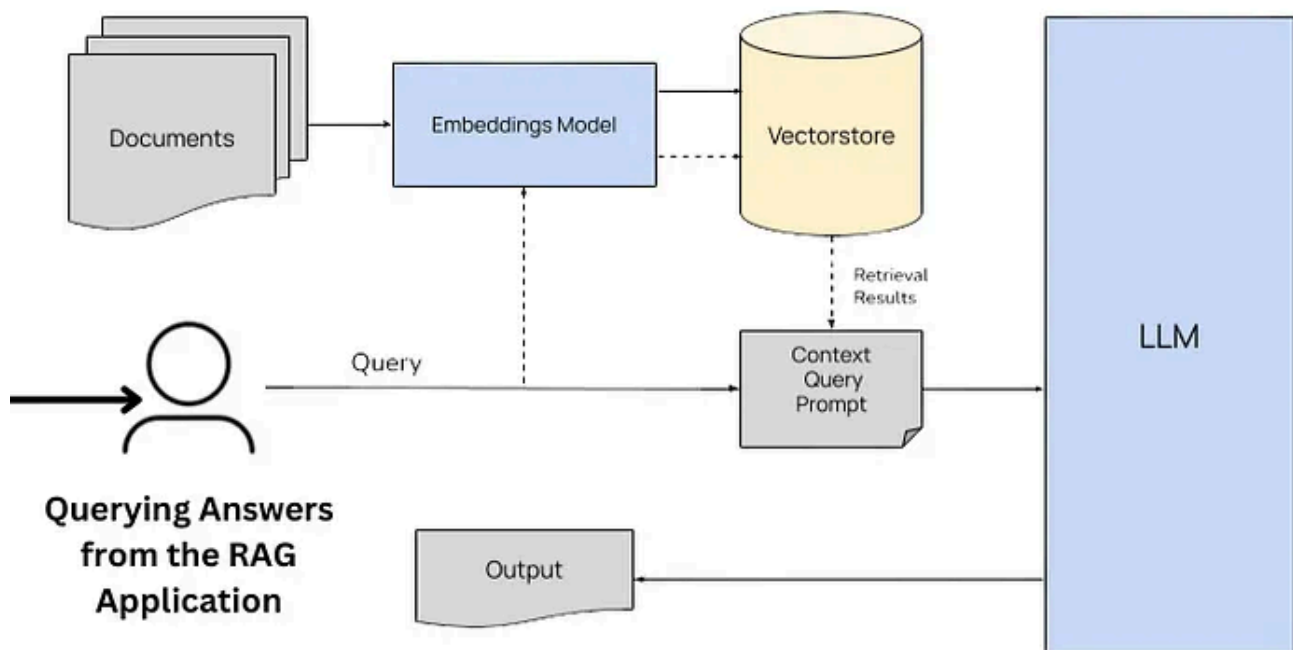
```
pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_tensors='pt',
    max_length=512,
    max_new_tokens=512,
    model_kwargs={"torch_dtype": torch.bfloat16},
    device="cuda"
)
```

Initialize the LLM with pipeline and model kwargs.

```
llm = HuggingFacePipeline(
    pipeline=pipe,
    model_kwargs={"temperature": 0.7, "max_length": 512},
)
```

Now it is time to use the vector store and the LLM for question-answering retrieval.

6. Querying Answers from the RAG Application



Step 6: Querying Answers from the RAG Application

The final step is to generate the answers using both the vector store and the LLM. First, we will generate embeddings to the input query or question retrieve the context from the vector store, and feed this to the LLM to generate the answers:

```

def ask_question(question):

    # Get the embedding for the question
    question_embedding = embeddings.encode(doc)

    # Search for similar vectors
    res = index.query(vector=question_embedding, top_k=5, include_metadata=True)

    # Collect the results in a context
    context = "\n".join([r.metadata['text'] for r in res])

    prompt = f"Question:{question}\n\nContext: {context}"

    # Generate the answer using the LLM
    answer = llm(prompt)

    # Return the generated answer
    return answer[0]['generated_text']
  
```

The RAG pipeline is now ready and we can pass an input query and observe the output and how it performs.

```
ask_question("Write an educational story for young children.")
```

Once upon a time, in a cozy little village nestled between rolling hills and green meadows, there lived a curious kitten named Whiskers. Whiskers loved to explore every nook and cranny of the village, from the bustling marketplace to the quiet corners where flowers bloomed. One sunny morning, as Whiskers trotted down the cobblestone path, he spotted something shimmering in the distance. With his whiskers twitching in excitement, he scampered towards it, his little paws pitter-pattering on the ground. To his delight, he found a shiny object peeking out from beneath a bush — a beautiful, colorful kite! With a twinkle in his eye, Whiskers decided to take the kite on an adventure. He tugged at the string, and the kite soared into the sky, dancing gracefully with the gentle breeze. Whiskers giggled with joy as he watched the kite soar higher and higher, painting the sky with its vibrant colors.

If you like the article and would like to support me, make sure to:

- 🙌 Clap for the story (50 claps) to help this article be featured
- Subscribe to [To Data & Beyond](#) Newsletter
- Follow me on [Medium](#)
- 📰 View more content on my [medium profile](#)
- 🔔 Follow Me: [LinkedIn](#) | [Youtube](#) | [GitHub](#) | [Twitter](#)

Subscribe to my newsletter To Data & Beyond to get full and early access to my articles:

To Data & Beyond | Youssef Hosni | Substack


Data Science, Machine Learning, AI, and what is beyond them. Click to read To Data & Beyond, by Youssef Hosni, a...

youssefh.substack.com



Are you looking to start a career in data science and AI and do not know how? I offer data science mentoring sessions and long-term career mentoring:

- Mentoring sessions: <https://lnkd.in/dXeg3KPW>
- Long-term mentoring: <https://lnkd.in/dtdUYBrM>

Let's have a 1:1 call



- 1:1 Free Mentorship
- Data Science Career Planning
- Data Science Project Review

 topmate.io/youssef_hosni

Data Science

AI

LLm

Deep Learning

Machine Learning

More from the list: "RAG"

Curated by Geraldo Checon