# 🦜 LangChain + Streamlit 🔥 + Llama 🦙: Bringing Conversational AI to Your Local Machine

generative ai, chatgpt, how to use llm offline, large language models, how to make offline chatbot, document question answering using language models, machine learning, artificial intelligence, using llama on local machine, use language models on local machine

*By Afaque Umer on June 23rd, 2023*

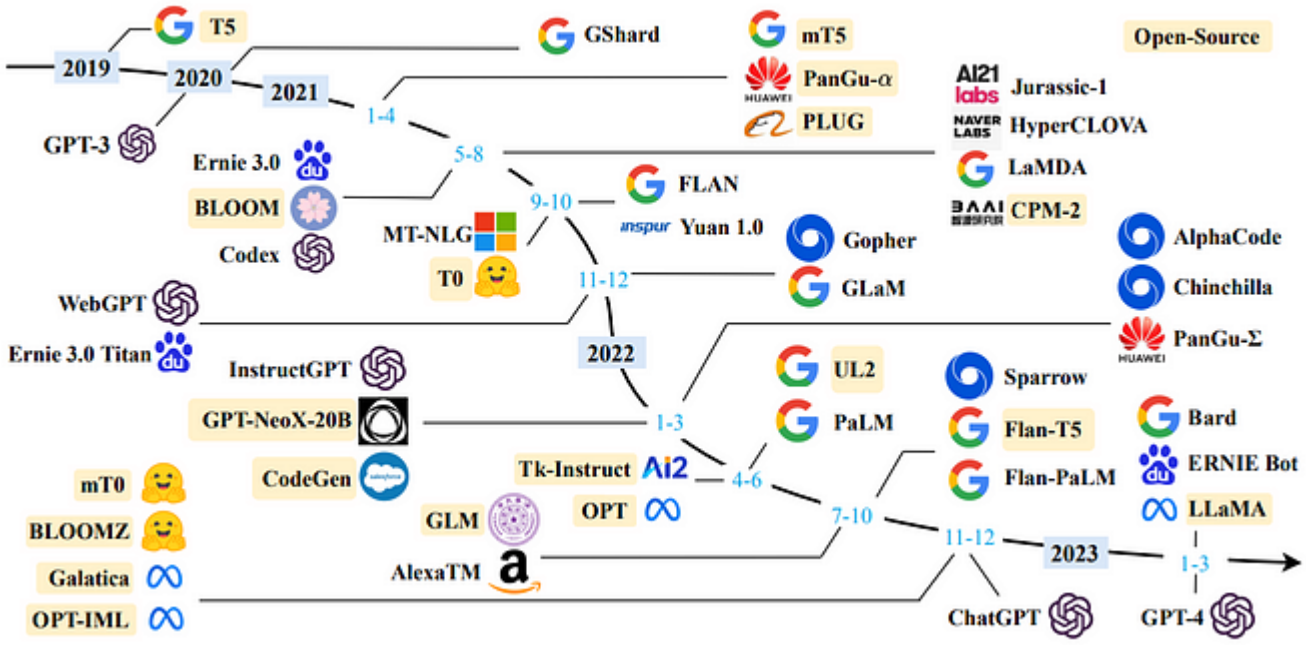
**Artificial Intelligence** **Data Science** **LLMs** **Langchain**
**Large Language Models** **Llama** **Machine Learning** **Python**

Integrating Open Source LLMs and LangChain for Free ***Generative Question Answering*** (No API Key required)

In the past few months, ***Large Language Models (LLMs)*** have gained significant attention, capturing the interest of developers across the planet. These models have created exciting prospects, especially for developers working on chatbots, personal assistants, and content creation. The possibilities that LLMs bring to the table have sparked a wave of enthusiasm in the Developer | AI | NLP community.

## What are LLMs?

Large Language Models (LLMs) refer to machine learning models capable of producing text that closely resembles human language and comprehending prompts in a natural manner. These models undergo training using extensive datasets comprising books, articles, websites, and other sources. By analyzing statistical patterns within the data, LLMs predict the most probable words or phrases that should follow a given input.




**A timeline of LLMs in recent years: A Survey of Large Language Models**

By utilizing Large Language Models (LLMs), we can incorporate domain-specific data to address inquiries effectively. This becomes especially advantageous when dealing with information that was not accessible to the model during its initial training, such as a company's internal documentation or knowledge repository.

The architecture employed for this purpose is known as ***Retrieval Augmentation Generation*** or, less commonly, ***Generative Question Answering.***

## What is LangChain 🦜🔗 ?

LangChain is an impressive and freely available framework meticulously crafted to empower developers in creating applications fueled by the might of language models, particularly large language models (LLMs).

LangChain revolutionizes the development process of a wide range of applications, including chatbots, Generative Question-Answering (GQA), and summarization. By

seamlessly **chaining** 🔗 together components sourced from multiple modules, LangChain enables the creation of exceptional applications tailored around the power of LLMs.

**Read More: Official Documentation**

## Motivation?

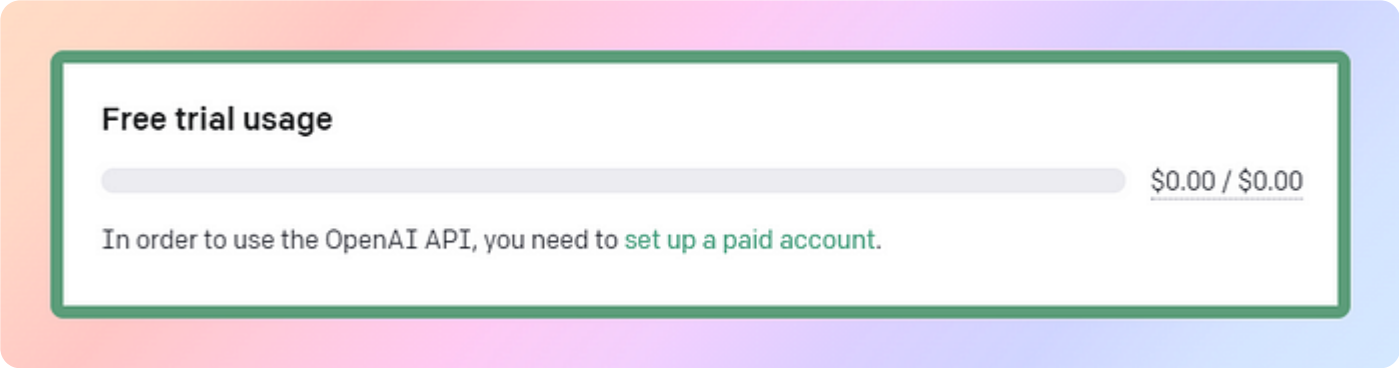



**Image By Author**

In this article, I will demonstrate the process of creating your own Document Assistant from the ground up, utilizing LLaMA 7b and Langchain, an open-source library specifically developed for seamless integration with LLMs.

Here is an overview of the blog's structure, outlining the specific sections that will provide a detailed breakdown of the process:

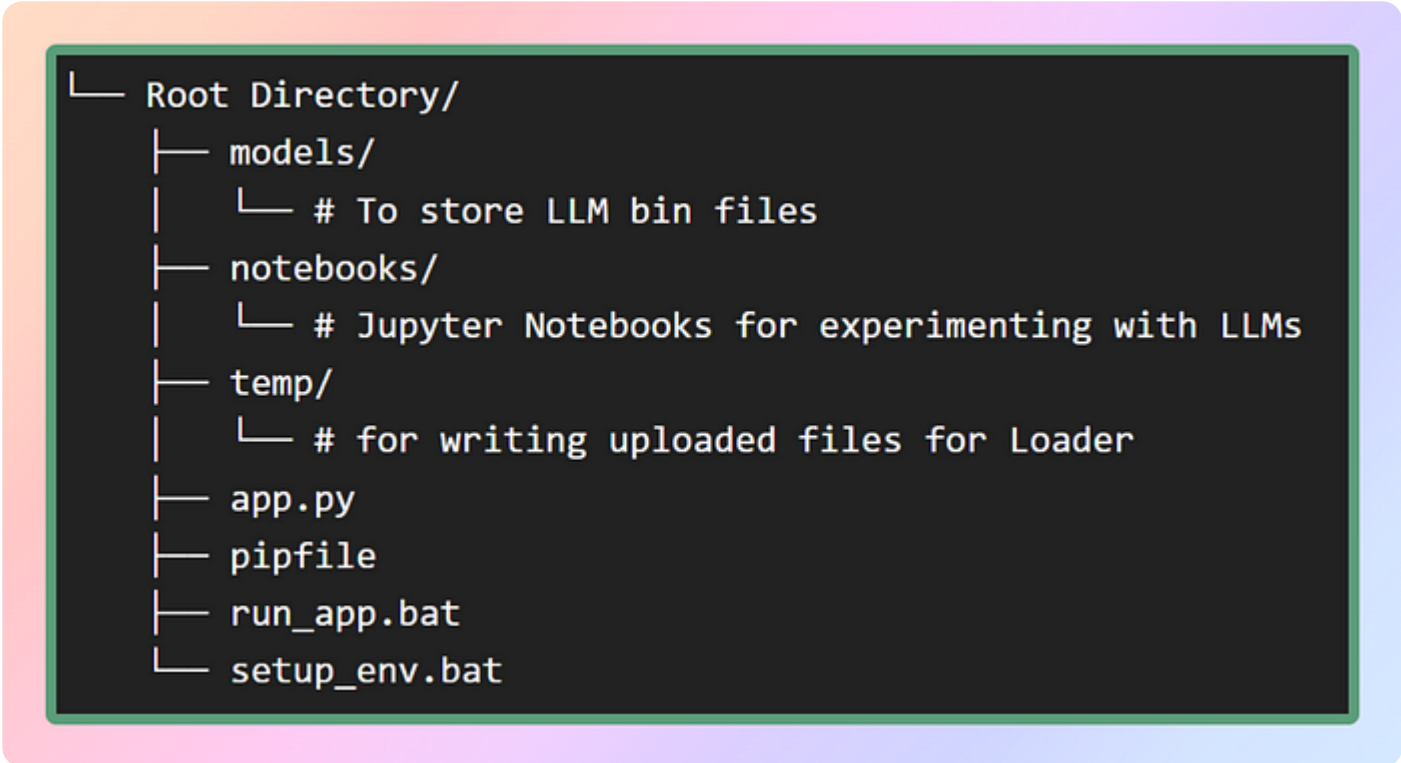
- `**Setting up the virtual environment and creating file structure**`
- `**Getting LLM on your local machine**`
- `**Integrating LLM with LangChain and customizing PromptTemplate**`
- `**Document Retrieval and Answer Generation**`

- **Building application using Streamlit**

## Section 1: Setting Up the Virtual Environment and Creating File Structure

Setting up a virtual environment provides a controlled and isolated environment for running the application, ensuring that its dependencies are separate from other system-wide packages. This approach simplifies the management of dependencies and helps maintain consistency across different environments.

To set up the virtual environment for this application, I will provide the pip file in my GitHub repository. First, let's create the necessary file structure as depicted in the figure. Alternatively, you can simply clone the repository to obtain the required files.




### Image By Author: File Structure

Inside the models' folder, we will store the LLMs that we will download, while the pip file will be located in the root directory.

To create the virtual environment and install all the dependencies within it, we can use the **`pipenv install`**command from the same directory or simply run ⚙️**`setup_env.bat`** batch file, It will install all the dependencies from the **`pipfile`**. This will ensure that all the necessary packages and libraries are installed in the virtual environment. Once the dependencies are successfully installed, we can proceed to the next step, which involves downloading the desired models. Here is the repo 👇

## Section 2: Getting LLaMA on your local machine

**What is LLaMA?**

LLaMA is a new large language model designed by Meta AI, which is Facebook's parent company. With a diverse collection of models ranging from 7 billion to 65 billion parameters, LLaMA stands out as one of the most comprehensive language models available. On February 24th, 2023, Meta released the LLaMA model to the public, demonstrating their dedication to open science.

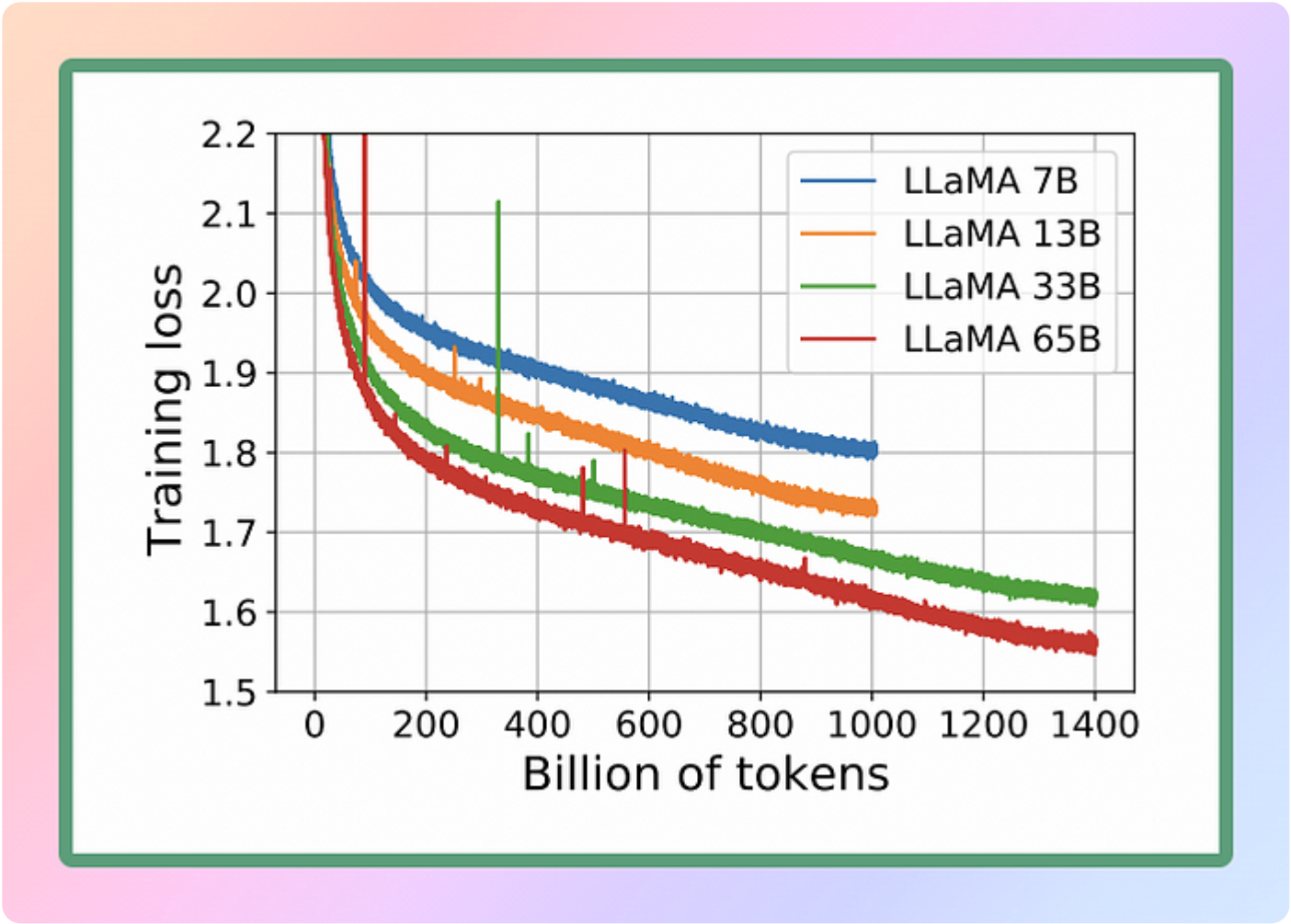



**Image Source: LLaMA**

Considering the remarkable capabilities of LLaMA, we have chosen to utilize this powerful language model for our purposes. Specifically, we will be employing the smallest version of LLaMA, known as LLaMA 7B. Even at this reduced size, LLaMA 7B offers significant language processing capabilities, allowing us to achieve our desired outcomes efficiently and effectively.

***Official Research Paper*** : `[**LLaMA: Open and Efficient Foundation Language Models**]`
`(https://research.facebook.com/publications/llama-open-and-efficient-foundation-language-models/)`

To execute the LLM on a local CPU, we need a local model in GGML format. Several methods can achieve this, but the simplest approach is to download the bin file directly from the **Hugging Face Models repository** 🤗. In our case, we will download the Llama 7B model. These models are open-source and freely available for download.

If you're looking to save time and effort, don't worry — I've got you covered. Here's the direct link for you to download the models ⏬. Simply download any version of it and then move the file into the models directory within our root directory. This way, you'll have the model conveniently accessible for your usage.

#### What is GGML? Why GGML? How GGML? LLaMA CPP??

GGML is a Tensor library for machine learning, it is just a C++ library that allows you to run LLMs on just the CPU or CPU + GPU. It defines a binary format for distributing large language models (LLMs). GGML makes use of a technique called ***quantization*** that allows for large language models to run on consumer hardware.

#### now what is Quantization?

LLM weights are floating point (decimal) numbers. Just like it requires more space to represent a large integer (e.g. 1000) compared to a small integer (e.g. 1), it requires more space to represent a high-precision floating point number (e.g. 0.0001) compared to a low-precision floating number (e.g. 0.1). The process of ***quantizing*** a large language model involves reducing the precision with which weights are represented in order to reduce the resources required to use the model. GGML supports a number of different quantization strategies (e.g. 4-bit, 5-bit, and 8-bit quantization), each of which offers different trade-offs between efficiency and performance.

| Model | Original size | Quantized size (4-bit) |
|---|---|---|
| 7B | 13 GB | 3.9 GB |
| 13B | 24 GB | 7.8 GB |
| 30B | 60 GB | 19.5 GB |
| 65B | 120 GB | 38.5 GB |

**Quantized Size of Llama**

To effectively use the models, it is essential to consider the memory and disk requirements. Since the models are currently loaded entirely into memory, you will need sufficient disk space to store them and enough RAM to load them during execution. When it comes to the 65B model, even after quantization, it is recommended to have at least 40 gigabytes of RAM available. It's worth noting that the memory and disk requirements are currently equivalent.

Quantization plays a crucial role in managing these resource demands. Unless you have access to exceptional computational resources 🤑🤑🤑

By reducing the precision of the model's parameters and optimizing memory usage, quantization enables the models to be utilized on more modest hardware configurations. This ensures that running the models remains feasible and efficient for a wider range of setups.

**how do we use it in Python if it's a C++ library?**

That's where Python bindings come into play. Binding refers to the process of creating a bridge or interface between two languages for us python and C++. We will use **llama-cpp-python**which is a Python binding for **llama.cpp** which acts as an Inference of the LLaMA model in pure C/C++. The main goal of **llama.cpp** is to run the LLaMA model using 4-bit integer quantization. This integration allows us to effectively utilize the LLaMA model, leveraging the advantages of C/C++ implementation and the benefits of 4-bit integer quantization

🚀


**Supported Models by llama.cpp : Source**

With the GGML model prepared and all our dependencies in place (thanks to the pipfile), it's time to embark on our journey with LangChain. But before diving into the exciting world of LangChain, let's kick things off with the customary **"Hello World"**

ritual — a tradition we follow whenever exploring a new language or framework, after all, LLM is also a language model 😄.


**Image By Author: Interaction with LLM on CPU**

*Voilà !!!* We have successfully executed our first LLM on the CPU, completely offline and in a fully randomized fashion(you can play with the hyper param **temperature**).

With this exciting milestone accomplished 🎯, we are now ready to embark on our primary objective: question answering of custom text using the LangChain framework.

## Section 3: Getting Started with LLM — LangChain Integration 🤝

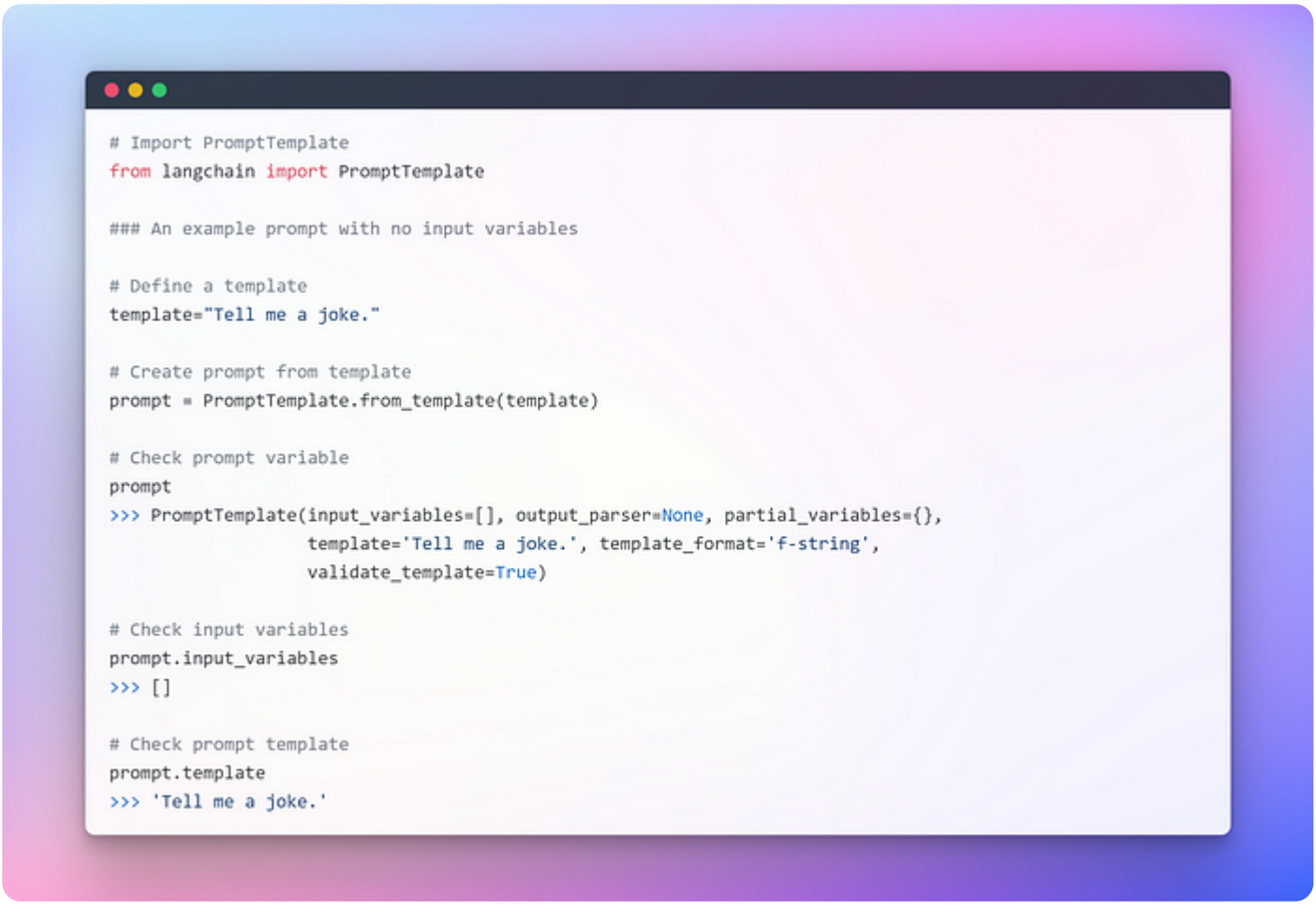
In the last section, we initialized LLM using llama cpp. Now, let's leverage the LangChain framework to develop applications using LLMs. The primary interface through which you can interact with them is through text. As an oversimplification, a lot of models are ⬇️ **text in, text out** ⬆️. Therefore, a lot of the interfaces in LangChain are centered around the text.

**The Rise of Prompt Engineering** 📈

In the ever-evolving field of programming a fascinating paradigm has emerged: **Prompting**. Prompting involves providing specific input to a language model to elicit a desired response. This innovative approach allows us to shape the output of the model based on the input we provide.

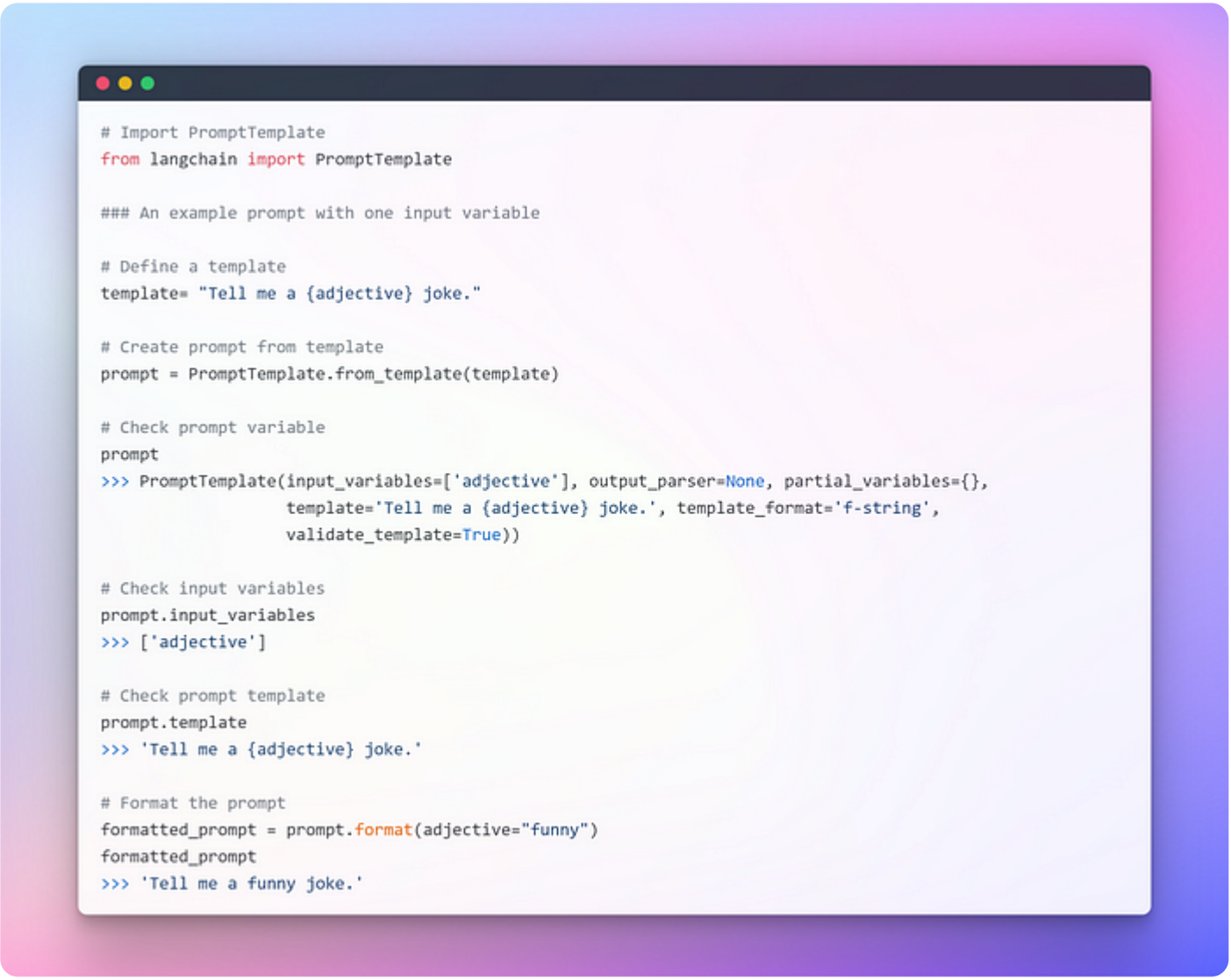
It’s remarkable how the nuances in the way we phrase a prompt can significantly impact the nature and substance of the model’s response. The outcome may vary fundamentally based on the wording, highlighting the importance of careful consideration when formulating prompts.

For providing seamless interaction with LLMs, LangChain provides several classes and functions to make constructing and working with prompts easy using a prompt template. It is a reproducible way to generate a prompt. It contains a text string **the template**, that can take in a set of parameters from the end user and generates a prompt. Let’s take a few examples.



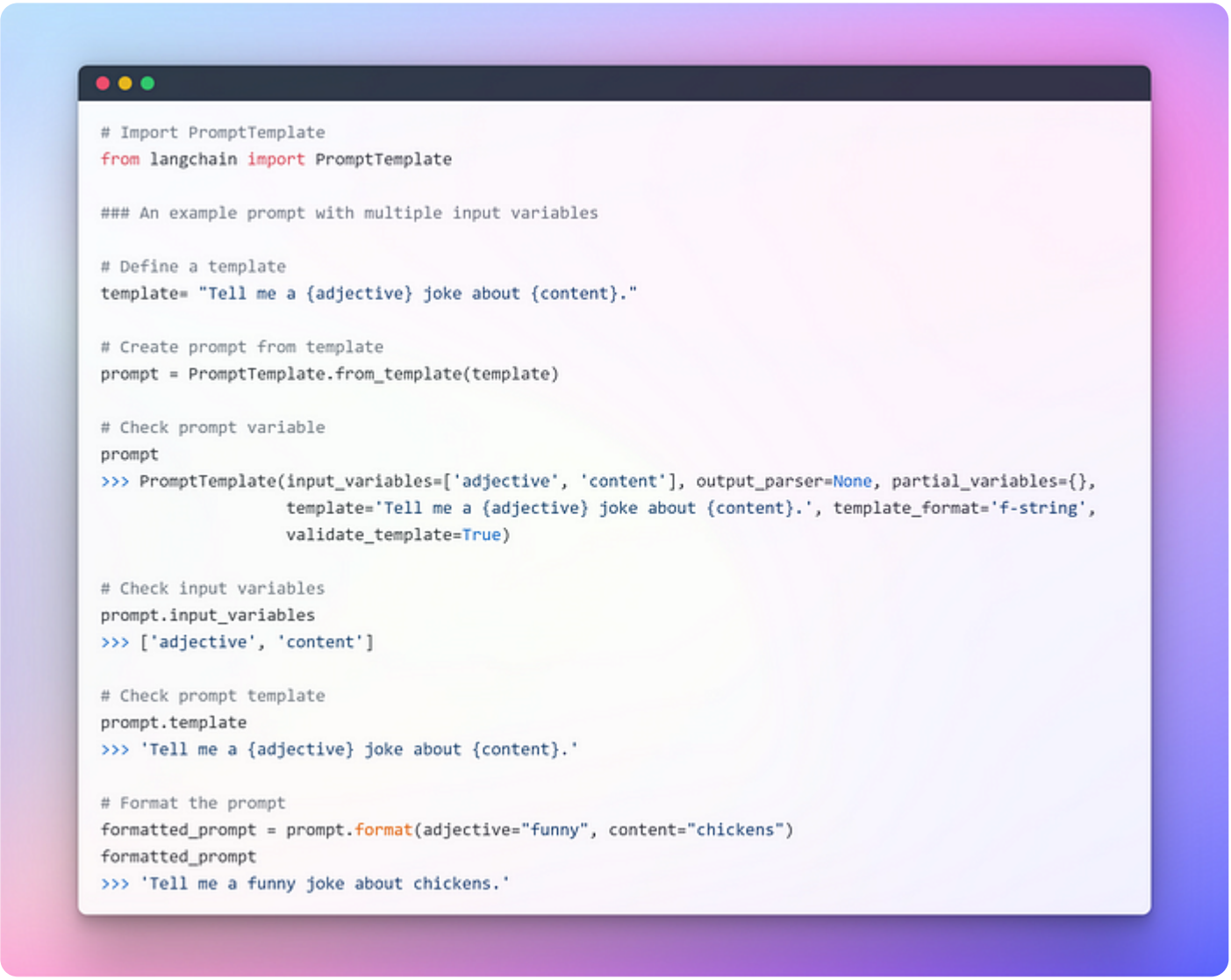
```
# Import PromptTemplate
from langchain import PromptTemplate

### An example prompt with no input variables

# Define a template
template="Tell me a joke."

# Create prompt from template
prompt = PromptTemplate.from_template(template)

# Check prompt variable
prompt
>>> PromptTemplate(input_variables=[], output_parser=None, partial_variables={},
                   template='Tell me a joke.', template_format='f-string',
                   validate_template=True)

# Check input variables
prompt.input_variables
>>> []

# Check prompt template
prompt.template
>>> 'Tell me a joke.'
```

## Image By Author: Prompt with no Input Variables



```
# Import PromptTemplate
from langchain import PromptTemplate

### An example prompt with one input variable

# Define a template
template= "Tell me a {adjective} joke."

# Create prompt from template
prompt = PromptTemplate.from_template(template)

# Check prompt variable
prompt
>>> PromptTemplate(input_variables=['adjective'], output_parser=None, partial_variables={},
                   template='Tell me a {adjective} joke.', template_format='f-string',
                   validate_template=True))

# Check input variables
prompt.input_variables
>>> ['adjective']

# Check prompt template
prompt.template
>>> 'Tell me a {adjective} joke.'

# Format the prompt
formatted_prompt = prompt.format(adjective="funny")
formatted_prompt
>>> 'Tell me a funny joke.'
```

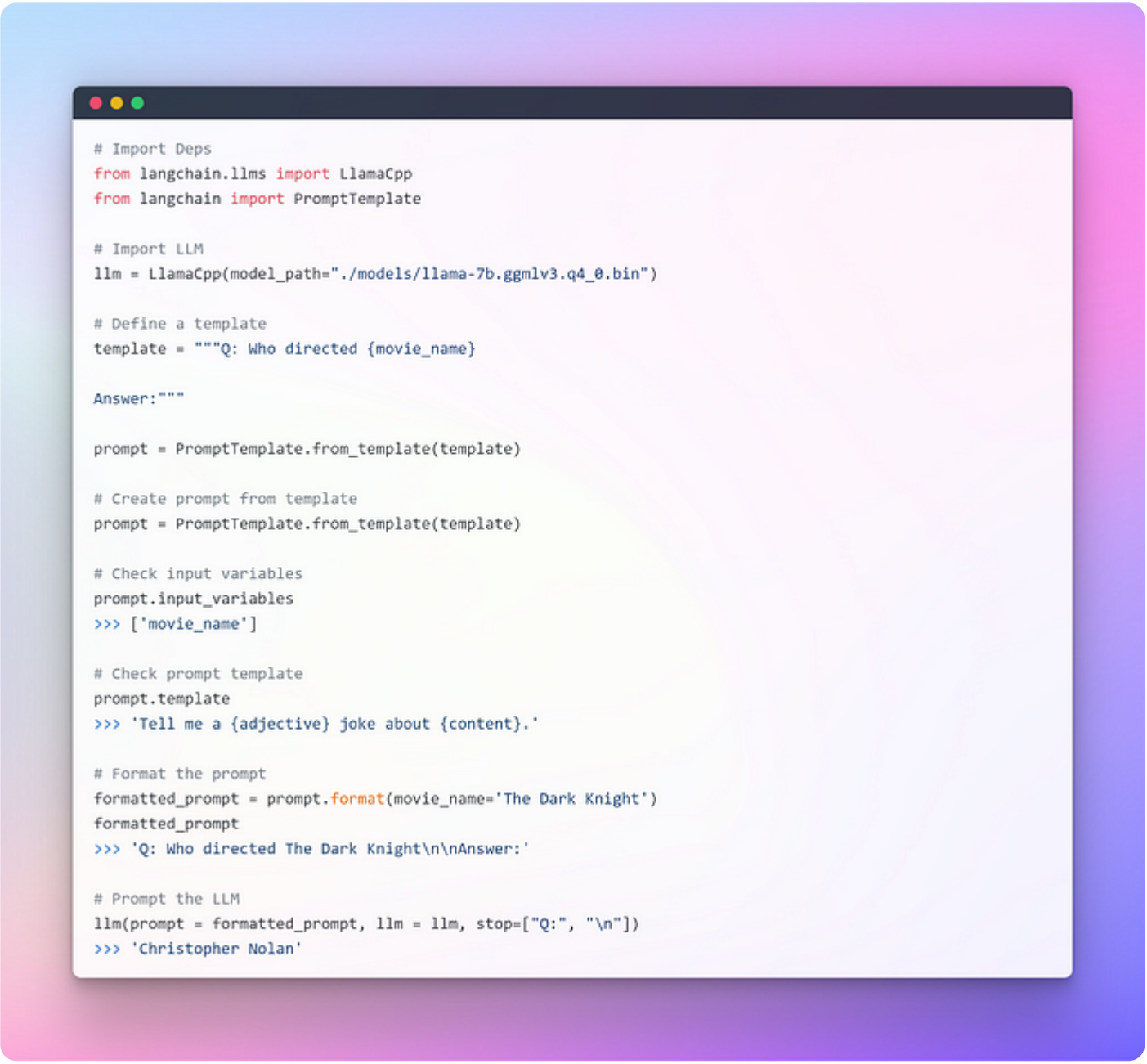
## Image By Author: Prompt with one Input Variables



```
# Import PromptTemplate
from langchain import PromptTemplate

### An example prompt with multiple input variables

# Define a template
template= "Tell me a {adjective} joke about {content}."

# Create prompt from template
prompt = PromptTemplate.from_template(template)

# Check prompt variable
prompt
>>> PromptTemplate(input_variables=['adjective', 'content'], output_parser=None, partial_variables={},
                   template='Tell me a {adjective} joke about {content}.', template_format='f-string',
                   validate_template=True)

# Check input variables
prompt.input_variables
>>> ['adjective', 'content']

# Check prompt template
prompt.template
>>> 'Tell me a {adjective} joke about {content}.'

# Format the prompt
formatted_prompt = prompt.format(adjective="funny", content="chickens")
formatted_prompt
>>> 'Tell me a funny joke about chickens.'
```

**Image By Author: Prompt with multiple Input Variables**

I hope that the previous explanation has provided a clearer grasp of the concept of prompting. Now, let’s proceed to prompt the LLM.



```
# Import Deps
from langchain.llms import LlamaCpp
from langchain import PromptTemplate

# Import LLM
llm = LlamaCpp(model_path="./models/llama-7b.ggmlv3.q4_0.bin")

# Define a template
template = """Q: Who directed {movie_name}

Answer:"""

prompt = PromptTemplate.from_template(template)

# Create prompt from template
prompt = PromptTemplate.from_template(template)

# Check input variables
prompt.input_variables
>>> ['movie_name']

# Check prompt template
prompt.template
>>> 'Tell me a {adjective} joke about {content}.'

# Format the prompt
formatted_prompt = prompt.format(movie_name='The Dark Knight')
formatted_prompt
>>> 'Q: Who directed The Dark Knight\n\nAnswer:'

# Prompt the LLM
llm(prompt = formatted_prompt, llm = llm, stop=["Q:", "\n"])
>>> 'Christopher Nolan'
```

### Image By Author: Prompting through Langchain LLM

This worked perfectly fine but this ain't the optimum utilisation of LangChain. So far we have used individual components. We took the prompt template formatted it, then took the llm, and then passed those params inside llm to generate the answer. Using an LLM in isolation is fine for simple applications, but more complex applications require chaining LLMs — either with each other or with other components.

LangChain provides the Chain interface for such **chained** 🔗 applications. We define a Chain very generically as a sequence of calls to components, which can include other chains. Chains allow us to combine multiple components together to create a single, coherent application. For example, we can create a chain that takes user input, formats it with a Prompt Template, and then passes the formatted response to an LLM. We can build more complex chains by combining multiple chains together, or by combining chains with other components.

To understand one let's create a very simple **chain** 🔗 that will take user input, format the prompt with it, and then send it to the LLM using the above individual components that we've already created.

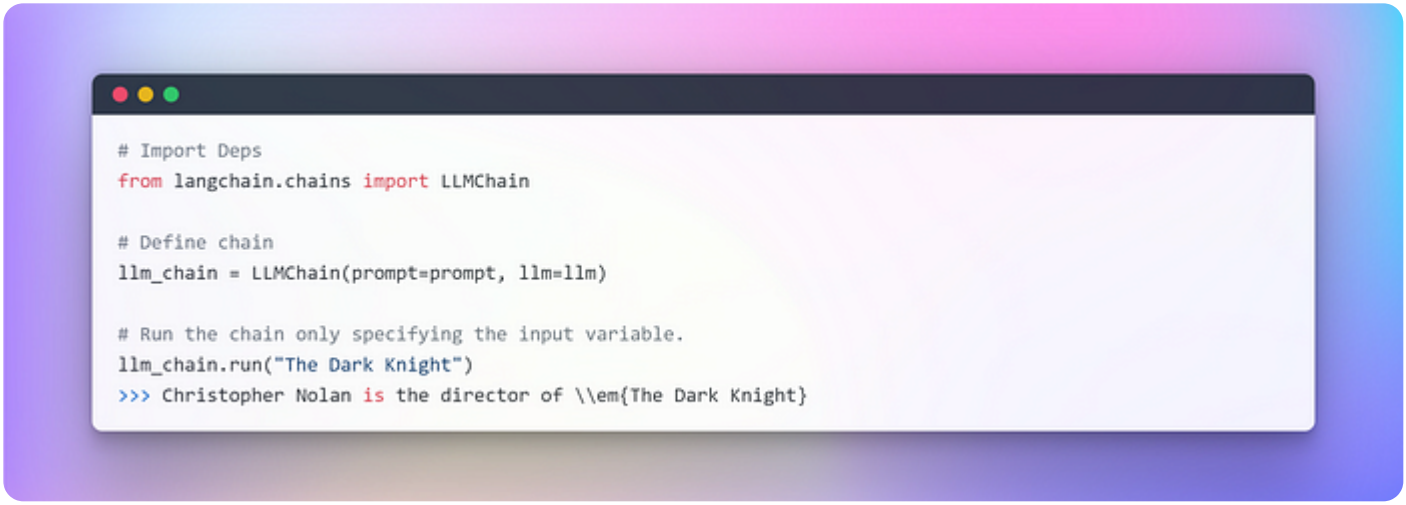


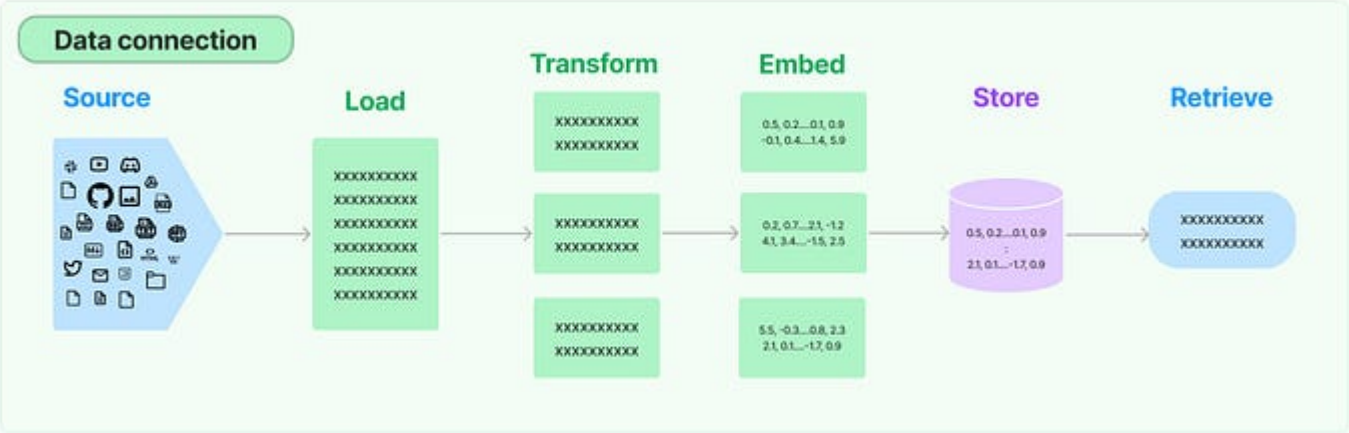

**Image By Author: Chaining in LangChain**



by utilizing a dictionary. That concludes this section. Now, let's dive into the main part where we'll incorporate external text as a retriever for question-answering purposes.

## Section 4: Generating Embeddings and Vectorstore for Question Answering

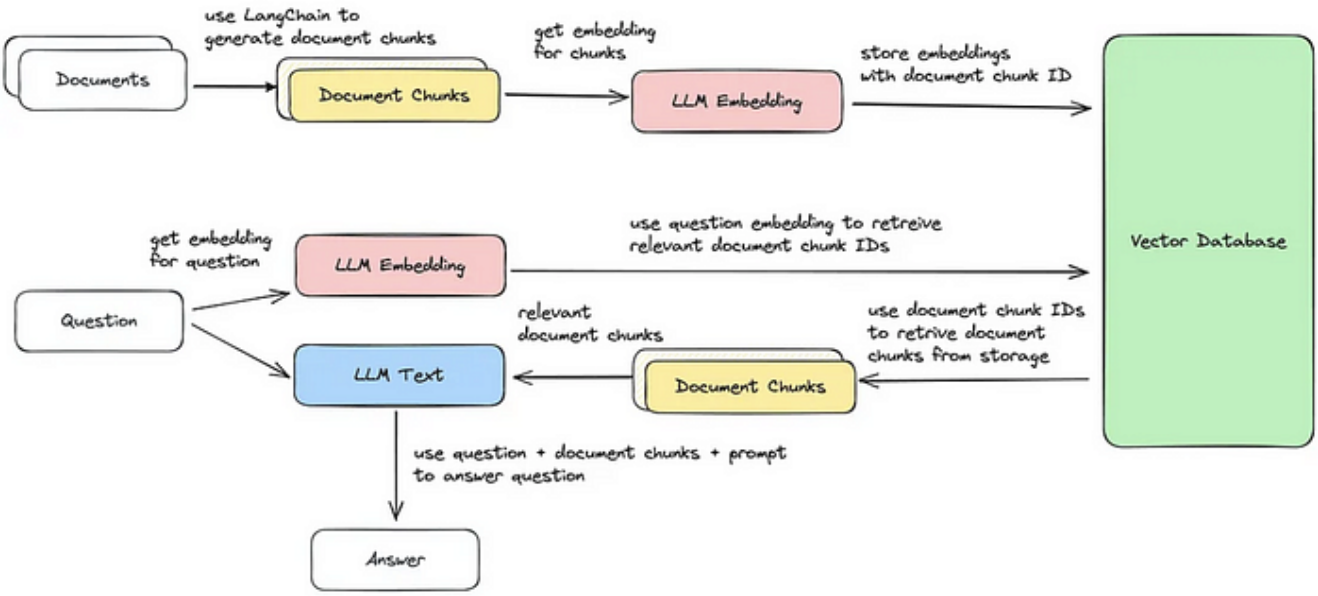
In numerous LLM applications, there is a need for user-specific data that isn't included in the model's training set. LangChain provides you with the essential components to load, transform, store, and query your data.




### Data Connection in LangChain: Source

The five stages are:

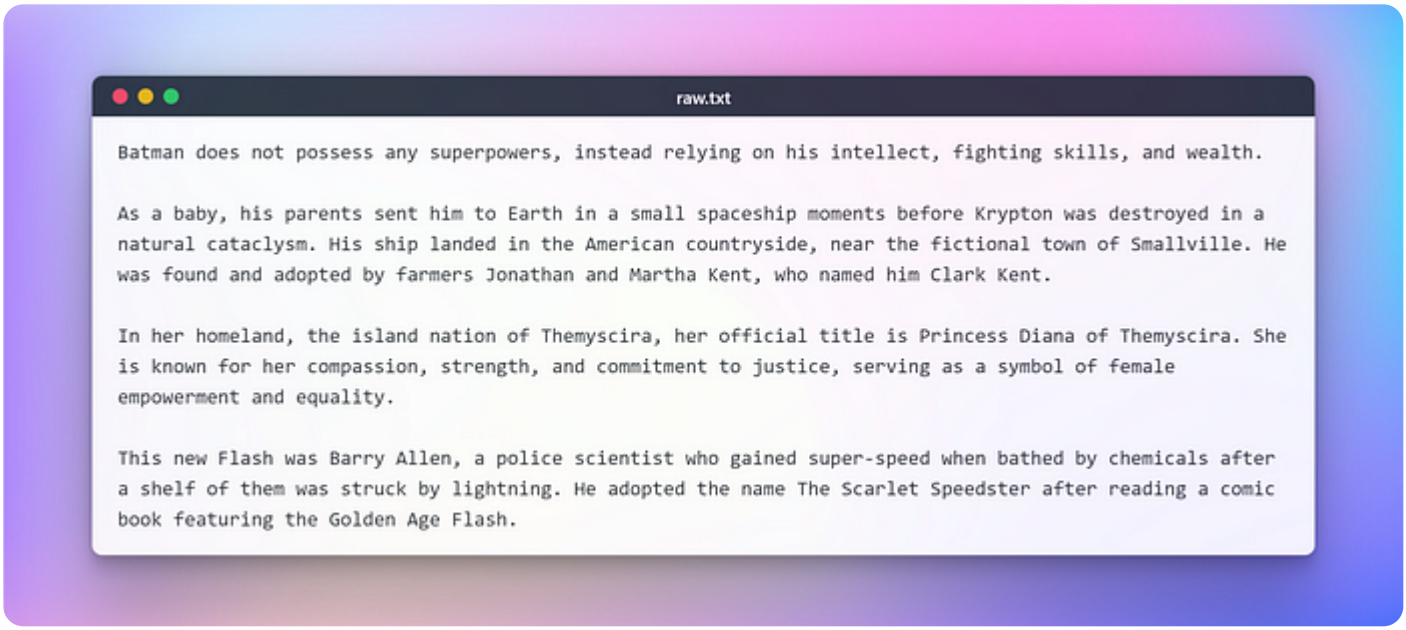
- **Document Loader:** It is used for loading data as documents.
- **Document Transformer:** It split the document into smaller chunks.
- **Embeddings:** It transforms the chunks into vector representations a.k.a embedding.
- **Vector Stores:** It is used to store the above chunk vectors in a vector database.
- **Retrievers:** It is used for retrieving a set/s of vector/s that are most similar to a query in a form of a vector that is embedded in the same Latent space.




## Document Retrieval / Question-Answering Cycle

Now, we will walk through each of the five steps to perform a retrieval of chunks of documents that are most similar to the query. Following that, we can generate an answer based on the retrieved vector chunk, as illustrated in the provided image.

However, before proceeding further, we will need to prepare a text for executing the aforementioned tasks. For the purpose of this fictitious test, I have copied a text from Wikipedia regarding some popular DC Superheroes. Here is the text:




## Image By Author: Raw Text for Testing

#### Loading & Transforming Documents

To begin, let's create a document object. In this example, we'll utilize the text loader. However, Lang chain offers support for multiple documents, so depending on your specific document, you can employ different loaders. Next, we'll employ the **load** method to retrieve data and load it as documents from a preconfigured source.

Once the document is loaded, we can proceed with the transformation process by breaking it into smaller chunks. To achieve this, we'll utilize the TextSplitter. By default, the splitter separates the document at the '\n\n' separator. However, if you set the separator to null and define a specific chunk size, each chunk will be of that specified length. Consequently, the resulting list length will be equal to the length of the document divided by the chunk size. In summary, it will resemble something like this: **list length = length of doc / chunk size**. Let's walk the talk.

```
# Import Deps
from langchain.document_loaders import TextLoader
from langchain.text_splitter import CharacterTextSplitter

# Load Doc
loader = TextLoader('./docs/raw.txt')
docs = loader.load()

# Transform into chunks
text_splitter = CharacterTextSplitter(chunk_size=10, chunk_overlap=0)
texts = text_splitter.split_documents(docs)

# compare size of docs and texts
len(docs)
>>> 1
len(texts)
>>> 4

# visualize chunks and doc
print(doc)
>>> [Document(page_content='Batman does not possess any superpowers, instead relying on his intellect,
    fighting skills, and wealth.\n\nAs a baby, his parents sent him to Earth in a small spaceship
    moments before Krypton was destroyed in a natural cataclysm. His ship landed in the American
    countryside, near the fictional town of Smallville. He was found and adopted by farmers Jonathan
    and Martha Kent, who named him Clark Kent.\n\nIn her homeland, the island nation of Themyscira, her
    official title is Princess Diana of Themyscira. When blending into the society outside of her
    homeland, she sometimes adopts her civilian identity Diana Prince.\n\nThis new Flash was Barry
    Allen, a police scientist who gained super-speed when bathed by chemicals after a shelf of them was
    struck by lightning. He adopted the name The Scarlet Speedster after reading a comic book featuring
    the Golden Age Flash.', metadata={'source': './docs/raw.txt'})]

print(texts[0])
>>> page_content='Batman does not possess any superpowers, instead relying on his intellect, fighting
                  skills, and wealth.' metadata={'source': './docs/raw.txt'}
```

## Image By Author: Loading and Transforming Doc

#### Part of the journey is the Embeddings

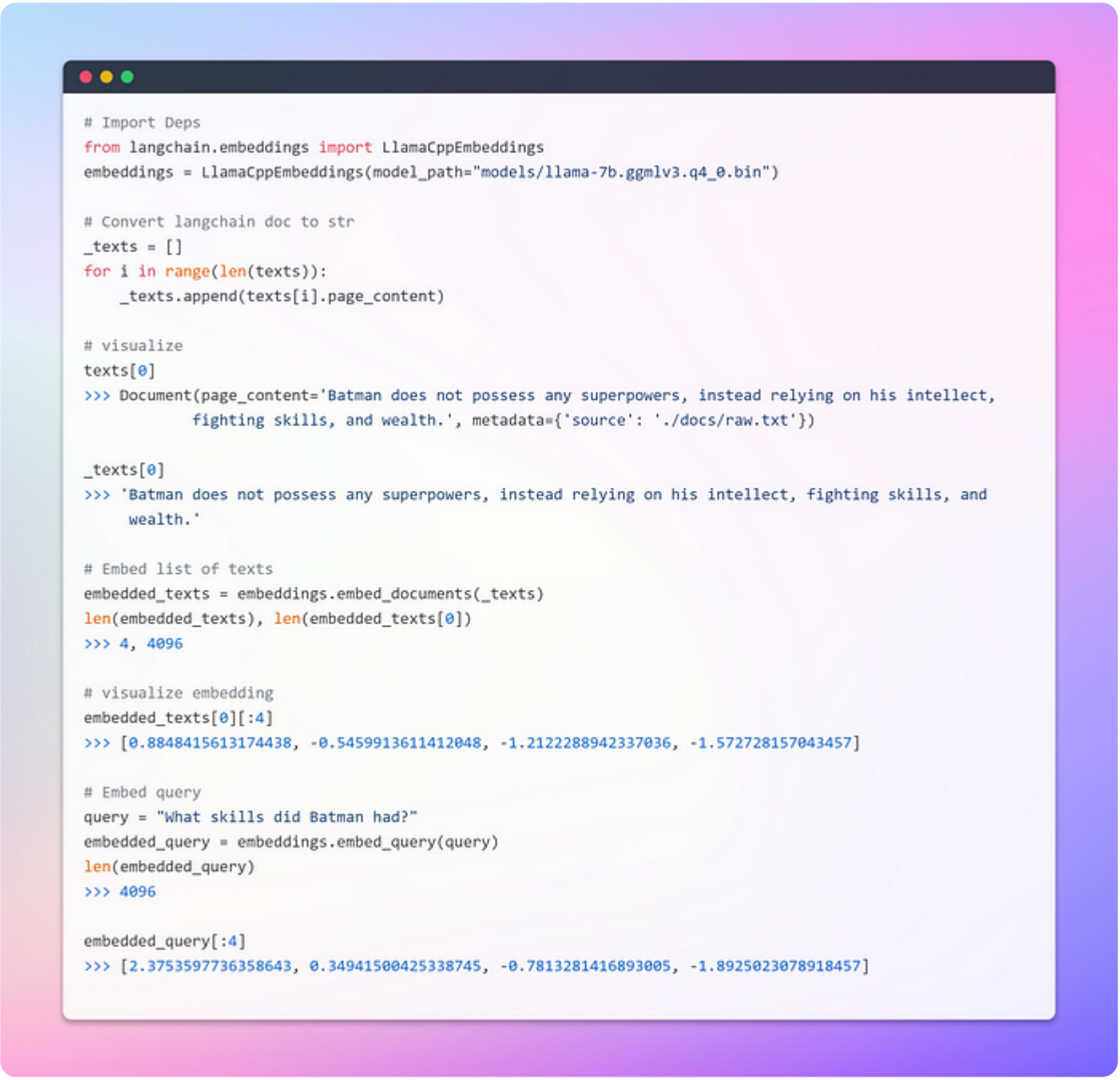
This is the most important step. Embeddings generate a vectorized portrayal of textual content. This has practical significance since it allows us to conceptualize text within a vector space.

Word embedding is simply a vector representation of a word, with the vector containing real numbers. Since languages typically contain at least tens of

thousands of words, simple binary word vectors can become impractical due to a high number of dimensions. Word embeddings solve this problem by providing dense representations of words in a low-dimensional vector space.

When we talk about retrieval, we refer to retrieving a set of vectors that are most similar to a query in a form of a vector that is embedded in the same Latent space.

The base Embeddings class in LangChain exposes two methods: one for embedding documents and one for embedding a query. The former takes as input multiple texts, while the latter takes a single text.



```
# Import Deps
from langchain.embeddings import LlamaCppEmbeddings
embeddings = LlamaCppEmbeddings(model_path="models/llama-7b.ggmlv3.q4_0.bin")

# Convert langchain doc to str
_texts = []
for i in range(len(texts)):
    _texts.append(texts[i].page_content)

# visualize
texts[0]
>>> Document(page_content='Batman does not possess any superpowers, instead relying on his intellect,
            fighting skills, and wealth.', metadata={'source': './docs/raw.txt'})

_texts[0]
>>> 'Batman does not possess any superpowers, instead relying on his intellect, fighting skills, and
     wealth.'

# Embed list of texts
embedded_texts = embeddings.embed_documents(_texts)
len(embedded_texts), len(embedded_texts[0])
>>> 4, 4096

# visualize embedding
embedded_texts[0][:4]
>>> [0.8848415613174438, -0.5459913611412048, -1.2122288942337036, -1.572728157043457]

# Embed query
query = "What skills did Batman had?"
embedded_query = embeddings.embed_query(query)
len(embedded_query)
>>> 4096

embedded_query[:4]
>>> [2.3753597736358643, 0.34941500425338745, -0.7813281416893005, -1.8925023078918457]
```

## Image By Author: Embeddings

For a comprehensive understanding of embeddings, I highly recommend delving into the fundamentals as they form the core of how neural networks handle textual data. I have extensively covered this topic in one of my blogs utilizing TensorFlow. Here is the link 👇

#### Creating Vector Store & Retrieving Docs

A vector store efficiently manages the storage of embedded data and facilitates vector search operations on your behalf. Embedding and storing the resulting embedding vectors is a prevalent method for storing and searching unstructured data. During query time, the unstructured query is also embedded, and the embedding vectors that exhibit the highest similarity to the embedded query are retrieved. This approach enables effective retrieval of relevant information from the vector store.

Here, we will utilize Chroma, an embedding database and vector store specifically crafted to simplify the development of AI applications incorporating embeddings. It offers a comprehensive suite of built-in tools and functionalities to facilitate your initial setup, all of which can be conveniently installed on your local machine by executing a simple `**pip install chromadb**` command.

```
# Import Deps
from langchain.vectorstores import Chroma

# Create a Chroma vectorstore from a list of documents.
db = Chroma.from_documents(texts, embeddings)

# Perform similary search with the query over db
query = "Who is an orphan here"
docs = db.similarity_search(query, k=1)
docs
>>> [Document(page_content='As a baby, his parents sent him to Earth in a small spaceship moments before
    Krypton was destroyed in a natural cataclysm. His ship landed in the American countryside, near the
    fictional town of Smallville. He was found and adopted by farmers Jonathan and Martha Kent, who
    named him Clark Kent.', metadata={'source': './docs/raw.txt'})]

# Search for documents using query vector
query = "Who is an orphan here"
query_vector = embeddings.embed_query(query)
docs = db.similarity_search_by_vector(query_vector, k=1)
docs
>>> [Document(page_content='As a baby, his parents sent him to Earth in a small spaceship moments before
    Krypton was destroyed in a natural cataclysm. His ship landed in the American countryside, near the
    fictional town of Smallville. He was found and adopted by farmers Jonathan and Martha Kent, who
    named him Clark Kent.', metadata={'source': './docs/raw.txt'})]
```

**Image By Author: Creating Vector Store**

Up until now, we've witnessed the remarkable capability of embeddings and vector stores in retrieving relevant chunks from extensive document collections. Now, the moment has come to present this retrieved chunk as a context alongside our query, to the LLM. With a flick of its magical wand, we shall beseech the LLM to generate an answer based on the information that we provided to it. The important part is the prompt structure.

However, it is crucial to emphasize the significance of a well-structured prompt. By formulating a well-crafted prompt, we can mitigate the potential for the LLM to engage in **hallucination** — wherein it might invent facts when faced with uncertainty.

Without prolonging the wait any further, let us now proceed to the final phase and discover if our LLM is capable of generating a compelling answer. The time has come to witness the culmination of our efforts and unveil the outcome. Here we Goooooo ⚡

```
# Import Deps
from langchain.chains import LLMChain
from langchain.prompts import PromptTemplate

# Craft a prompt template that works best for your LLM
template = """Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an answer.
{context}
Question: {question}
Answer:"""

# context will be the similar doc and question will be query
prompt = PromptTemplate.from_template(template)
prompt.input_variables
>>> ['context', 'question']

# prepare context from query
query = "Who is an orphan here?"

similar_doc = db.similarity_search(query, k=1)
context = similar_doc[0].page_content
print(context)
>> 'As a baby, his parents sent him to Earth in a small spaceship moments before Krypton was destroyed
   in a natural cataclysm. His ship landed in the American countryside, near the fictional town of
   Smallville. He was found and adopted by farmers Jonathan and Martha Kent, who named him Clark Kent.'

# Use LLM to Generate Answer from the context
query_llm = LLMChain(llm=llm, prompt=prompt)
response = query_llm.run({"context": context, "question": query})
print(response)
>>> Clark Kent
```

### Image By Author: Q/A with the Doc

This is the moment we've been waiting for! We've accomplished it! 👏 👏We have just built our very own question-answering bot 🤖utilizing the LLM running locally. ⚡ ⚡

## Section 5: Chain 🔗 All using Streamlit 🔥

This section is entirely optional since it doesn't serve as a comprehensive guide to Streamlit. I won't delve deep into this part; instead, I'll present a basic application that allows users to upload any text document. They will then have the option to ask questions through text input. Behind the scenes, the functionality will remain consistent with what we covered in the previous section.



However, there is a caveat when it comes to file uploads in Streamlit. To prevent potential out-of-memory errors, particularly considering the memory-intensive nature of LLMs, I'll simply read the document and write it to the temporary folder within our file structure, naming it `**raw.txt**`. This way, regardless of the document's original name, Textloader will seamlessly process it in the future.

Currently, the app is designed for text files, but you can adapt it for PDFs, CSVs, or other formats. The underlying concept remains the same since LLMs are primarily designed for text input and output. Additionally, you can experiment with different LLMs supported by the Llama C++ bindings.

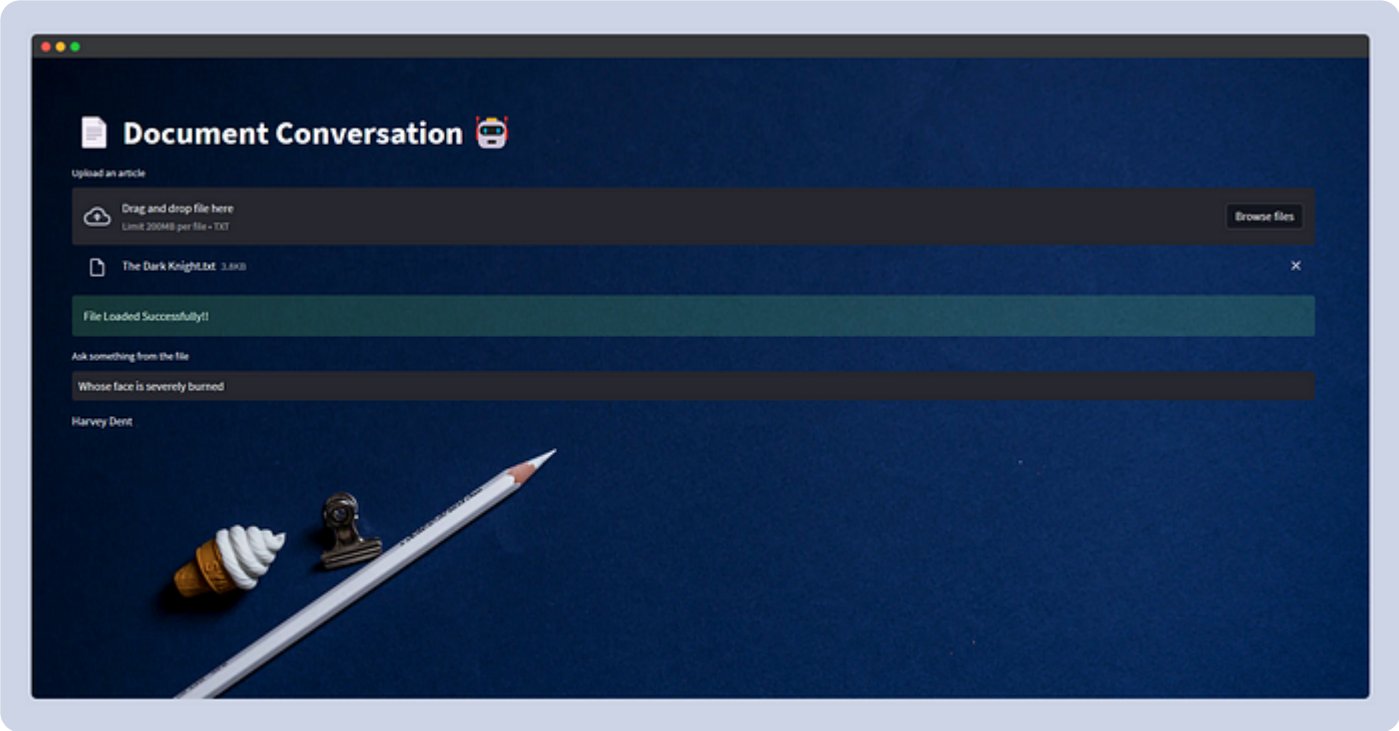
Without delving further into intricate details, I present the code for the app. Feel free to customize it to suit your specific use case.

```
# Bring in deps
import streamlit as st
from langchain.llms import LlamaCpp
from langchain.embeddings import LlamaCppEmbeddings
from langchain.prompts import PromptTemplate
from langchain.chains import LLMChain
from langchain.document_loaders import TextLoader
from langchain.text_splitter import
CharacterTextSplitter
from langchain.vectorstores import Chroma
```

```
# Customize the layout
st.set_page_config(page_title="DOCAI",
page_icon="🤖", layout="wide", )
st.markdown(f"""
             <style>
             .stApp {{background-image:
url("https://images.unsplash.com/photo-1509537257950-
20f875b03669?ixlib=rb-
4.0.3&ixid=M3wxMjA3fDB8MHxwaG90by1wYWdlfHx8fGVufDB8fHx
8fA%3D%3D&auto=format&fit=crop&w=1469&q=80");
                    background-attachment: fixed;
                    background-size: cover}}
          </style>
          """, unsafe_allow_html=True)

# function for writing uploaded file in temp
def write_text_file(content, file_path):
    try:
        with open(file_path, 'w') as file:
            file.write(content)
        return True
    except Exception as e:
        print(f"Error occurred while writing the
file: {e}")
        return False

# set prompt template
prompt_template = """Use the following pieces of
context to answer the question at the end. If you
don't know the answer, just say that you don't know,
don't try to make up an answer.
{context}
Question: {question}
Answer:"""
prompt = PromptTemplate(template=prompt_template,
input_variables=["context", "question"])

# initialize hte LLM & Embeddings
llm = LlamaCpp(model_path="./models/llama-
7b.ggmlv3.q4_0.bin")
embeddings =
LlamaCppEmbeddings(model_path="models/llama-
7b.ggmlv3.q4_0.bin")
llm_chain = LLMChain(llm=llm, prompt=prompt)

st.title("📄 Document Conversation 🤖")
uploaded_file = st.file_uploader("Upload an article",
type="txt")
```

```
if uploaded_file is not None:
    content = uploaded_file.read().decode('utf-8')
    # st.write(content)
    file_path = "temp/file.txt"
    write_text_file(content, file_path)

    loader = TextLoader(file_path)
    docs = loader.load()
    text_splitter =
CharacterTextSplitter(chunk_size=100, chunk_overlap=0)
    texts = text_splitter.split_documents(docs)
    db = Chroma.from_documents(texts, embeddings)
    st.success("File Loaded Successfully!!")

    # Query through LLM
    question = st.text_input("Ask something from the
file", placeholder="Find something similar to:
....this.... in the text?", disabled=not
uploaded_file,)
    if question:
        similar_doc = db.similarity_search(question,
k=1)
        context = similar_doc[0].page_content
        query_llm = LLMChain(llm=llm, prompt=prompt)
        response = query_llm.run({"context": context,
"question": question})
        st.write(response)
```

Here’s what the streamlit app will look like 🔥 .




This time I fed the plot of **The Dark Knight** copied from Wiki and just asked **Whose face is severely burnt?** and the LLM replied — **Harvey Dent**.

All right, all right, all right! With that, we come to the end of this blog.

I hope you enjoyed this article! and found it informative and engaging. You can **follow me Afaque Umer** for **more** such articles.

I will try to bring up more M**achine learning/Data science concepts** and will try to break down fancy-sounding terms and concepts into simpler ones.

All right, all right, all right! Goodbye 👋

**Thanks for reading 🙏 Keep learning 🧠 Keep Sharing 🤝 Stay Awesome 🤘**

# Continue Learning

### 8 Data Structures Every Python Programmer Should Know

algorithms data structures python

### The Best Free and Paid VPN Providers

A review of the most popular VPN providers in 2023.

cybersecurity privacy virtual private network vpn vpn providers

### How to Migrate an existing React Web App to Desktop App with Electron

desktop electron javascript react web

### The Advanced Way to Style with Styled Components

A more in-depth look at the power of styled-components

css css-in-js javascript react styled-components

### The Cost of the Cloud: Here's What You Need to Know about AWS Pricing

AWS Cloud

## Can You Transcribe Audio Using Python?

audio python