# Exploring LangChain: A Practical Approach to Language Models and Retrieval-Augmented Generation (RAG)

LangChain is a powerful framework for building applications that incorporate large language models (LLMs). It simplifies the process of embedding LLMs into complex workflows, enabling the creation of conversational agents, knowledge retrieval systems, automated pipelines, and other AI-driven applications.

At its core, LangChain follows a modular design that allows developers to build “chains,” or sequences of actions, with customizable components like prompt templates, model settings, response parsing, and memory management. It also supports integration with external data sources such as document databases, search indices, APIs, and more—a feature commonly referred to as Retrieval-Augmented Generation (RAG). This flexibility empowers developers to create tailored solutions for diverse tasks, from customer support bots to data analysis tools that extract insights from extensive datasets.

With a growing suite of tools and agents, LangChain is a top choice for developers aiming to leverage LLMs for dynamic, data-driven, and interactive applications. It offers extensive customization options and is actively maintained to support the latest advancements in LLMs and AI.

In this article, I’ll guide you through the basics of using LangChain and its components. You'll learn how to combine different modules to create functional applications, including a RAG application for querying private documents using LLMs.

## A Basic LangChain Example

The easiest way to get started with LangChain is to begin with a simple example. First, let’s install the following libraries using the **pip** command:

!pip install langchain

!pip install langchain-openai

For this example, you’ll be using LLMs from OpenAI, so you need to apply for an OpenAI API key and then save the API key in an environment variable:

import os

# replace with your own API Key

os.environ['OPENAI\_API\_KEY'] = "OpenAI API Key"

Do remember that OpenAI operates with a pay-for-use model, where costs are typically based on usage, like the number of tokens processed in language model queries. Go to <https://platform.openai.com/docs/overview> to sign up for an OpenAI account.

### Components In LangChain

In a LangChain application, components are connected or "chained" to create complex workflows for natural language processing. Each component in the chain serves a specific purpose, like prompting the model, managing memory, or processing outputs, and they pass information to each other to enable more sophisticated applications. By chaining these components, you can build systems that not only generate responses but also retrieve information, maintain conversational context, summarize content, and much more. This modular approach allows flexibility, letting you create pipelines that can adapt to various tasks and data inputs based on the needs of your application.

In this basic example, you’ll use the following components:

* **PromptTemplate**: This component helps create a structured template for the prompt. It allows you to specify placeholders that can be dynamically filled with specific inputs (like a question or context) each time the prompt is used. This makes it easier to standardize prompts while customizing them for each query.
* **ChatOpenAI**: This component interfaces with OpenAI's chat models, enabling the generation of responses based on the provided prompt and any contextual information. It acts as the core of the application, where the language model generates responses to the user's inputs.
* **StrOutputParser**: This component processes the raw output from the model into a usable format, such as extracting only the text content. It simplifies the response so that it can be easily displayed or further processed in your application.

By chaining these components, you can build a streamlined flow from prompt creation to response parsing, providing a solid foundation for more advanced LangChain applications. **Figure 1** shows how these components are chained together.



**Figure 1:** Chaining all the components in a LangChain application

Let’s create the first component: **PromptTemplate**:

from langchain import PromptTemplate

template = '''

Question: {question}

Answer: '''

prompt = PromptTemplate(

 template = template,

 input\_variables = ['question']

)

Basically, the **PromptTemplate** contains a string template that specifies the structure of the prompt, containing a Question field where **{question}** acts as a placeholder. When the template is used, this placeholder will be replaced with the actual question input.

The next component to create is the **ChatOpenAI**:

from langchain\_openai import ChatOpenAI

model = ChatOpenAI(model="gpt-4o-mini")

Here, you’re making use of the “gpt-4o-mini” model from OpenAI.

The third component is the **StrOutputParser**:

from langchain\_core.output\_parsers \

import StrOutputParser

output\_parser = StrOutputParser()

The **StrOutputParser** is used to handle the output from your model and parse it as a straightforward string. This is useful when working with responses that don’t require complex parsing or structuring. **StrOutputParser** will ensure that the model's output is returned as a raw string.

### Chaining the Components

Finally, you can now combine your components into a single chain by linking the **PromptTemplate**, **ChatOpenAI** model, and **StrOutputParser**. This chaining approach allows a streamlined pipeline where you can input a question, have it processed through each component, and receive a parsed response.

# create the chain

chain = prompt | model | output\_parser

In LangChain, the | operator is used to combine multiple components (such as PromptTemplate, ChatOpenAI, and StrOutputParser) into a chain. This "piping" operator allows you to create a seamless workflow where the output of one component is automatically passed as the input to the next.

### Invoking the Chain

To use the chain to answer a question, call its **invoke()** method and pass in a dictionary containing the **question** key and setting its value to the question you’re asking:

chain.invoke({"question": "Who is Steve Jobs?"})

The **invoke()** method in LangChain returns the final processed output after it flows through each component in the chain. You will see something like the following:

[AI Response]

Steve Jobs was an American entrepreneur, inventor, and business magnate best known as the co-founder of Apple Inc. He was born on February 24, 1955, and passed away on October 5, 2011. Jobs played a crucial role in the development of revolutionary products such as the Macintosh computer, iPod, iPhone, and iPad. He was known for his visionary approach to technology and design, as well as his emphasis on user experience. In addition to his work at Apple, Jobs was also the CEO of Pixar Animation Studios and played a significant role in the production of acclaimed films like "Toy Story." His leadership style and focus on innovation have left a lasting impact on the technology industry and popular culture.

This invoke() method simplifies the interaction with your LangChain application by handling all components in sequence and directly providing the final answer.

### Maintaining Conversations with Memory

If you've used ChatGPT before, you know it can handle follow-up questions seamlessly. For example, after asking, "Who is Steve Jobs?" you might follow up with, "What are some of the companies he founded?" ChatGPT understands that "he" refers to Steve Jobs and can provide relevant information about the companies he founded. This ability to maintain context across questions is possible because ChatGPT uses memory to keep track of the conversation.

The LangChain application that you built earlier, however, doesn’t have memory of the previous conversation. To prove that, ask a follow-up question:

chain.invoke({"question":

 "What company did he found?"})

And you will get the following response from the model:

[AI Response]

Could you please provide more context or specify who you are referring to? This will help me give you a more accurate answer.

To maintain a conversation with the model, you can use the **ConversationBufferMemory** component in LangChain. The **ConversationBufferMemory** component helps store the ongoing conversation’s context, allowing the model to remember previous inputs and responses. This memory buffer enables the model to refer back to earlier parts of the conversation, making follow-up questions and references more coherent.

To maintain a conversation effectively, you should modify the prompt template to include two placeholders: one for the conversation history and one for the current question. This structure allows the model to consider the entire context of the conversation when generating a response:

# Define the prompt template

template = '''

Previous conversation:

{history}

Question: {question}

Answer: '''

# Create the PromptTemplate with history

prompt = PromptTemplate(

 template = template,

 input\_variables = ['history', 'question']

)

To store the history of the conversation, create an instance of the **ConversationBufferMemory** class:

# Set up conversational memory

memory = ConversationBufferMemory()

To retrieve the history of the conversation whenever you ask the model a question, you can use the following statement:

memory.load\_memory\_variables({})["history"]}

Here’s how the above statement works:

* **memory.load\_memory\_variables({})**: This method retrieves the current memory variables stored in the **ConversationBufferMemory**. By passing an empty dictionary, you are requesting all memory variables without any filters.
* **["history"]**: This accesses the specific history variable from the retrieved memory. It provides the entire context of the conversation up to that point.

So now when you ask a question, you create a dictionary with two keys: **question** and **history**:

response = chain.invoke(

 {"question" : question,

 "history" :

 memory.load\_memory\_variables({})["history"]} )

print(response)

Essentially, whenever you ask a question, you are also passing back the history of the conversation to the model so that it can provide the context for the current question.

When the model returns a response, you should save the context to the **ConversationBufferMemory** instance using the **save\_context()** method. This method allows you to store both the question and the answer, thereby updating the conversation history. Here’s how you can do this:

memory.save\_context(

 {"question": question},

 {"answer": response} )

**Listing 1** shows the complete application that can maintain a conversation with the model.

Two questions were asked. The model prints out the following output:

[AI Response]

Steve Jobs was an American entrepreneur, inventor, and business magnate best known for co-founding Apple Inc. in 1976. He played a key role in the development of revolutionary products such as the Macintosh computer, iPod, iPhone, and iPad, which helped to transform the technology and consumer electronics industries. Jobs was known for his visionary leadership, design-focused approach, and emphasis on user experience. He was also the CEO of Pixar Animation Studios, contributing to the success of animated films like "Toy Story." Jobs passed away on October 5, 2011, but his legacy continues to influence technology and design today.

Steve Jobs co-founded Apple Inc. in 1976.

The **ConversationBufferMemory** class provides two ways to access the chat history:

* **memory.load\_memory\_variables({})["history"]** provides a formatted and concise view of the conversation history, ideal for use in prompts.
* **memory.chat\_memory.messages** gives direct access to the raw messages in a structured format, suitable for deeper inspection or manipulation.

Let’s examine the result returned by **memory.chat\_memory.messages**:

[HumanMessage(content='Who is Steve Jobs?'),

 AIMessage(content='Steve Jobs was an American

entrepreneur, inventor, and business magnate

best known for co-founding Apple Inc. in 1976.

He played a key role in the development of

revolutionary products such as the Macintosh

computer, iPod, iPhone, and iPad, which helped to

transform the technology and consumer electronics

industries. Jobs was known for his visionary

leadership, design-focused approach, and emphasis

on user experience. He was also the CEO of Pixar

Animation Studios, contributing to the success of

animated films like "Toy Story." Jobs passed away

on October 5, 2011, but his legacy continues to

influence technology and design today.'),

HumanMessage(content='What company did he found?'),

AIMessage(content='Steve Jobs co-founded Apple Inc.

in 1976.')]

Observe that each question contains two objects: **HumanMessage** (question asked by the user) and **AIMessage** (response from the model).

### Sticking within the LLM Context Size

Although using a **ConversationBufferMemory** object to maintain an ongoing conversation by passing back the history can be effective, there is one potential problem: **memory overload** or **context length limitations**. Most language models, including those based on the GPT architecture, have a maximum token limit for the input they can process at one time. If the conversation history becomes too lengthy, you may exceed this token limit. Also, as the context grows, the computational load increases. This can lead to slower responses and increased resource consumption, affecting the performance of your application.

There are a couple of techniques to prevent context length limitations. Here are the two most common ways to resolve this issue:

* **Truncating the chat history**: Limit the conversation history to the most recent exchanges.
* **Summarizing the past interactions:** Instead of including the full conversations in the past, summarize them to keep the context.

With the first approach, you can send in only the most recent two messages by filtering the last two messages, like this:

response = chain.invoke(

 {"question": question,

 "history" :

 memory.chat\_memory.messages[-2 \* 2:]})

Remember, each conversation has two components **HumanMessage** and **AIMessage**. Hence you need to multiply by two in the above statement.

For the second approach, the idea is that once the conversation history becomes too lengthy, you should summarize the previous interactions into a more concise format. This helps manage memory usage and maintain a relevant context without overwhelming the model with excessive detail. **Listing 2** shows how you can make use of the Hugging Face transformers’ **pipeline** object to perform the summarization:

Note that you need to install the transformers library using the **pip** command:

!pip install transformers

You can now modify your program so that if the history contains more than four questions, summarize the history by calling the **summarize\_history()** function (see **Listing 3**).

You can now chat for as long as you want!

### Asking Multiple Questions

The **invoke()** method allows you to pass a question to the chain. Instead of passing this method a single dictionary, you can pass it a list of dictionaries if you want to ask multiple questions in one go. **Listing 4** shows how this is done.

Based on the questions, the chain returns the following result:

[AI Response]

1. The population of Singapore is approximately 5.6 million people as of 2023.

2. The question of whether the egg or chicken came first is a philosophical and scientific debate. From a biological perspective, it's generally accepted that the egg came first, as birds evolved from reptiles, which laid eggs long before chickens existed.

### Prompt for Language Translation

Up to this point, the examples have primarily focused on querying the language model (LLM) with questions. However, you can enhance the functionality of the prompt template to facilitate task-oriented requests. For instance, you can modify the prompt to instruct the LLM to perform specific tasks, such as translating a sentence from one language to another. By adjusting the prompt structure, you enable the model to understand the context of the task and respond accordingly, effectively broadening the scope of interactions beyond mere inquiries.

The example shown in **Listing 5** demonstrates how to create a translation chain, which integrates a prompt template, a language model, and an output parser to facilitate translating sentences between languages. The **PromptTemplate** is structured to take in the source language, target language, and the sentence to be translated. The code allows users to invoke the chain with specific input values, resulting in the desired translation. In this example, the English sentence "How are you?" is translated into Chinese.

The chain returns the following result: 'Translation: 你好吗？'.

### Exploring Alternatives to OpenAI LLMs

Until now, our focus has been on OpenAI's large language models (LLMs). Although these models deliver excellent results, they also involve operational expenses. A viable alternative to consider is leveraging models from Hugging Face, which can provide similar capabilities without the associated costs.

To make use of the Hugging Face models in LangChain, you need to install the following libraries:

!pip install langchain\_community

!pip install langchain-huggingface

**Listing 6** shows how you use the **HuggingFaceEndPoint** class to use the **tiiuae/falcon-7b-instruct** model to answer questions.

The tiiuae/falcon-7b-instruct model is a large language model developed by the Technology Innovation Institute, featuring approximately 7 billion parameters designed for instruction-based tasks. It excels in understanding and following instructions, making it suitable for various natural language processing applications such as answering questions, generating text, summarizing content, and engaging in dialogue.

Note that when you use the **HuggingFaceEndPoint** class, the inferencing is performed on Hugging Face’s server. You will see the following printout from the application:

\nSteve Jobs was an American entrepreneur who co-founded Apple Inc. and was instrumental in the creation of the personal computer and the development of the digital media industry. He is widely considered one of the most influential innovators of the 20th century.

## Implementing RAG with LangChain

Retrieval-Augmented Generation (RAG) is a robust technique in natural language processing that synergizes the retrieval of relevant information with the generation of contextually appropriate responses. This combination enhances tasks such as question answering, dialogue generation, and content creation, allowing organizations to deliver more accurate and pertinent answers to user queries. In the previous examples, the large language models (LLMs) used were pre-trained on a static dataset, which restricts their knowledge to the information available at the time of training. This limitation can hinder their ability to provide up-to-date or specific answers, especially when dealing with rapidly changing information or niche topics. RAG addresses this challenge by integrating real-time data retrieval, enabling models to access and incorporate fresh information into their responses.

In this section, I’ll guide you through the process of creating a RAG application using LangChain. In this example, I’ll provide a long paragraph of text as the input and leverage a large language model (LLM) to answer questions related to that text. This approach will demonstrate how RAG can enhance the model's ability to generate accurate and contextually relevant responses by combining the retrieval of information with generative capabilities. In a real-world scenario, this example could be expanded to handle documents stored in various formats, such as PDF, Word, or plain text. By integrating document loaders and retrieval mechanisms, the application could process and extract relevant information from these files, enabling the language model to answer questions based on a much broader range of sources. This extension makes the RAG approach especially useful for applications in knowledge management, customer support, and research, where information often exists in diverse document formats.

### Installing the Libraries

For this example, you need to install the following libraries:

!pip install langchain docarray tiktoken

Once the libraries are installed, import the relevant modules:

from langchain\_community.vectorstores import \

 DocArrayInMemorySearch

from langchain\_core.output\_parsers import \

 StrOutputParser

from langchain\_core.prompts import \

 ChatPromptTemplate

from langchain\_core.runnables import \

 RunnableParallel, RunnablePassthrough

from langchain\_openai import OpenAIEmbeddings

from langchain\_openai import ChatOpenAI

import os

You also need to ensure that your OpenAI API key is set:

# replace with your own API Key

os.environ['OPENAI\_API\_KEY'] = "OpenAI API Key"

### Defining the Text

Next, define a variable called **text** (see **Listing 7**) to store a paragraph that provides information about diabetes. You’ll use this paragraph as the reference content for the Retrieval-Augmented Generation (RAG) application, allowing the model to answer questions based on this specific text.

### Steps to Performing RAG

To get a large language model (LLM) to answer questions based on a specific document, follow these steps:

1. **Perform word vector embeddings on the document**: Word vector embeddings convert the document text into a numerical representation that captures the semantic meaning of the words and sentences. This allows the model to understand relationships between concepts in the document, making it easier to retrieve relevant information. Each sentence or passage is transformed into a vector, which helps locate information relevant to the question.
2. **Store the embeddings in a vector database**: After creating embeddings for each segment of the document, store them in a vector database (e.g., Pinecone, Weaviate, or FAISS). This database indexes the embeddings, enabling quick and efficient retrieval based on semantic similarity.
3. **Retrieve relevant document sections**: Search the vector database for sections of the document with embeddings similar to the question's embedding. This retrieves the most contextually relevant parts of the document.
4. **Pass retrieved sections to the LLM for answer generation**: Finally, pass the retrieved sections along with the question to the LLM. By doing this, the model can generate an informed answer based on the content of the document rather than relying solely on its pre-trained knowledge.

**Figure 2** shows the process.



**Figure 2:** Steps for performing RAG

### Chunking with Overlaps

Before performing word vector embeddings, it’s important to break down the documents into chunks because of:

1. **Context Size Limits**: LLMs and vector databases have constraints on the length of text they can process at once. Chunking ensures that each segment stays within these limits, preventing issues with model input size and vector database compatibility.
2. **Improved Semantic Representation**: Smaller, focused chunks allow for more precise embeddings that capture specific meanings, enhancing the accuracy of similarity searches. Larger segments may combine too many ideas, making it difficult for the model to retrieve the most relevant information.
3. **Efficient Retrieval**: Chunking enables a more targeted retrieval process. When a user asks a question, smaller segments can be selectively retrieved based on relevance. This makes retrieval faster and prevents overwhelming the model with unnecessary information.

**Figure 3** shows that a document is typically broken down into chunks with overlap, which involves dividing a block of text into segments that share a portion of their content. This method is particularly beneficial for maintaining context across adjacent chunks, ensuring that critical information is not lost during processing. By including overlapping sections, the model can better understand relationships between sentences and provide more accurate responses to queries, especially when dealing with complex topics that span multiple segments.



**Figure 3:** Document chunking with overlap

Now that you understand the process, let’s define a function named **split\_text\_with\_overlap()** that splits a block of text into a specific chunk size with a specific number of sentence to overlap (see **Listing 8**).

The block of text is now split into 10 chunks (see **Figure 4**).



**Figure 4:** The block of text is split into 10 chunks with overlaps

You’ll now use the **DocArrayInMemorySearch** class to store document embeddings in memory for efficient similarity search:

# creates an DocArrayInMemorySearch store and

# insert data

vectorstore = DocArrayInMemorySearch.from\_texts(

 text\_chunks,

 embedding = OpenAIEmbeddings(),

)

You’ll use the chunks that you have created and perform word vector embeddings using the **OpenAIEmbeddings** class.

OpenAIEmbeddings is a class provided by LangChain that allows you to generate vector embeddings for text using OpenAI's models. These embeddings are vector representations that capture the semantic meaning of the text, enabling efficient similarity searches, document retrieval, and other natural language processing (NLP) tasks where understanding the meaning of text is crucial.

### Creating a Retriever Object

You can now convert the vector store into a retriever object, which can be used to search and retrieve relevant documents based on a query:

retriever = vectorstore.as\_retriever()

A retriever object is a component designed to fetch relevant information from a dataset, document collection, or knowledge base based on a given query or context.

You can now create a LangChain application using the following components:

* RunnableParallel
* PromptTemplate
* ChatOpenAI
* StrOutputParser

**Listing 9** shows how the various components are created and then chained together.

The component of interest here is the **RunnableParallel** component. **RunnableParallel** is a class in LangChain that allows you to execute multiple tasks or operations in parallel:

setup\_and\_retrieval = RunnableParallel(

 {

 "context": retriever,

 "question": RunnablePassthrough()

 }

)

In this implementation, the **setup\_and\_retrieval** object is designed to handle two parallel tasks: retrieving context from a retriever and passing through a question without any modifications.

### Creating the Chain

Finally, the various components are chained together:

chain = setup\_and\_retrieval | \

 prompt | \

 model | \

 output\_parser

You can now start asking questions pertaining to the block of text:

chain.invoke('What is Type 2 diabetes?')

You will get the following response:

[AI Response]

Type 2 diabetes is often associated with insulin resistance and is more prevalent in adults, although it is increasingly observed in children and adolescents due to rising obesity rates. It is a chronic metabolic disorder characterized by high blood sugar levels, which can lead to serious health complications if not effectively managed. Risk factors for developing Type 2 diabetes include genetic predisposition, sedentary lifestyle, poor dietary choices, and obesity, particularly visceral fat that contributes to insulin resistance.

Here's another question:

chain.invoke('What causes diabetes?')

And you get the following response:

[AI Response]

Diabetes is caused by high blood sugar levels, which can result from various factors. For Type 1 diabetes, it is an autoimmune condition where the immune system attacks insulin-producing beta cells in the pancreas, leading to little or no insulin production. For Type 2 diabetes, it is often associated with insulin resistance and is influenced by risk factors such as genetic predisposition, sedentary lifestyle, poor dietary choices, and obesity, particularly visceral fat that contributes to insulin resistance.

### Changing the Embedding Model

For the RAG application you've developed so far, you've used OpenAI for the following components:

* **Word vector embeddings**: OpenAI's models have been employed to convert text documents into numerical representations (embeddings) that capture the semantic meaning of the text. This allows you to effectively compare and retrieve relevant information based on user queries.
* **Large language model (LLM)**: You've used OpenAI's LLM to generate responses and answer questions based on the retrieved context. This model leverages its training on vast amounts of text to provide coherent and contextually appropriate answers.

Although OpenAI’s models are efficient, using them can come with trade-offs, particularly concerning privacy. When you send data to OpenAI for processing—whether for generating embeddings or responses—there is a potential risk of exposing sensitive information. This is because of:

1. **Data transmission**: Queries and documents must be transmitted over the internet to OpenAI's servers, which could expose them to interception or unauthorized access.
2. **Data storage**: Depending on the terms of service, the data you send might be stored by OpenAI for training or improvement purposes, which raises concerns about how that data is used and who has access to it.
3. **Compliance**: Organizations handling sensitive information, especially in regulated industries, may face compliance challenges when using cloud-based solutions, as they need to ensure that they meet data protection regulations.

For those concerned about privacy, alternative solutions, such as local or self-hosted models (like those from Hugging Face), can be considered. These models allow you to maintain control over your data, ensuring that sensitive information remains within your own infrastructure.

In this section, you’ll replace OpenAI's models with those from Hugging Face. First, you’ll use the “BAAI/bge-small-en-v1.5” model for embedding:

from langchain.embeddings import \

 HuggingFaceEmbeddings

embedding\_model = \

HuggingFaceEmbeddings(

 model\_name="BAAI/bge-small-en-v1.5")

The model **BAAI/bge-small-en-v1.5** is a pre-trained language model developed by the BAAI (Beijing Academy of Artificial Intelligence). It’s part of the BGE (BERT-based generative embedding) series and is designed for various natural language processing tasks, including embedding generation.

Like the example, earlier, you’ll use the **DocArrayInMemorySearch** class to store document embeddings in memory:

from langchain\_community.vectorstores import \

 DocArrayInMemorySearch

vectorstore = DocArrayInMemorySearch.from\_texts(

 text\_chunks,

 embedding = embedding\_model,

)

retriever = vectorstore.as\_retriever()

Using the vector store created, you’ll create a retriever.

### Changing the LLM

Apart from using a model from Hugging Face for word vector embeddings, you’ll also use a large language model (LLM) from Hugging Face to perform the response generation. This allows you to keep the entire pipeline local or within the Hugging Face ecosystem, enhancing data privacy and reducing dependency on external APIs. **Listing 10** shows that you can make use of the **facebook/bart-large** model via the **pipeline** object in the **transformers** library:

# Load a Hugging Face pipeline for text generation

generator = pipeline('text2text-generation',

 model='facebook/bart-large',

 max\_length=500,

 device=device)

# Create a LangChain LLM wrapper

model = HuggingFacePipeline(pipeline=generator)

Also note that you can offload the processing to the GPU (for Windows; if you have a supported NVIDIA GPU) or MPS (if you have an Apple Silicon Mac):

# determine the device

if torch.backends.mps.is\_available():

 # for Apple Silicon Mac

 device = torch.device("mps")

else:

device = torch.device(

 "cuda" if torch.cuda.is\_available()

 else "cpu")

# Load a Hugging Face pipeline for text generation

generator = pipeline('text2text-generation',

 model='facebook/bart-large',

 max\_length=500,

 **device=device**)

Using the chain created, you can now ask a question where inference happens locally on your computer. This set up ensures that both the embedding retrieval and the large language model (LLM) processing occur on your own hardware, reducing reliance on external servers and improving data privacy:

chain.invoke('What is diabetes?')

You’ll get a response like the following:

[AI Response]

Human: Answer the question based only on the following context:[Document(page\_content='Diabetes mellitus is a chronic metabolic disorder characterized by high blood sugar levels, which can lead to serious health complications if not effectively managed. There are two primary types of diabetes: Type 1 diabetes, which is an autoimmune condition where the immune system mistakenly attacks insulin-producing beta cells in the pancreas, leading to little or no insulin production; and Type 2 diabetes, a type of diabetes that is more often associated with insulin resistance and is more prevalent in adults, though increasingly observed in children and adolescents due to rising obesity rates')]Question: What is diabetes? What is the most common cause of type 2 diabetes?Human: If you can answer this question, please do so in the following way: Document('Diabetes is a disease that affects the body's ability to produce and use insulin, the hormone responsible for regulating blood sugar. It is a condition that can be life-threatening if not properly managed'), Document('Type 2 diabetes is a serious condition')] Document(document('Diagnosis')]Document(document(document\_title='Type 2 Diabetes')Document(Document\_description)Document(doc\_title)Document\_content(document.doc\_content)Document-content(doc-content-1) Document\_content-2(doc)-content-3

The output from Hugging Face models can vary significantly based on several factors, such as model type, configuration, and input parameters. For example, generation models may produce different styles or lengths of responses based on settings like **temperature, max\_length, or top\_k/top\_p** sampling parameters. When using Hugging Face models in applications, you may need to tweak these settings or use output parsers to ensure consistency in responses, especially in tasks like question answering or summarization, where stable and contextually relevant outputs are important. I’ll leave this topic for another article.

## Summary

In this article, I walked you through the essential components and practical applications of the LangChain framework. It starts with a basic example to establish a foundation, then explores chaining components, managing conversation memory, and using LangChain's memory features to stay within LLM context limits. Additionally, it covers language translation prompts, alternative models beyond OpenAI, and a hands-on guide to implementing Retrieval-Augmented Generation (RAG) for document-based querying. With clear steps on chunking, creating retriever objects, and customizing embeddings, I hope this article provides you with a solid starting point for using LangChain in various NLP tasks.

Listing 1. Maintaining a conversation with the model

from langchain.memory import ConversationBufferMemory

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_openai import ChatOpenAI

from langchain import PromptTemplate

# Define the prompt template

template = '''

Previous conversation:

{history}

Question: {question}

Answer: '''

# Create the PromptTemplate with history

prompt = PromptTemplate(

 template = template,

 input\_variables = ['history', 'question']

)

# Set up conversational memory

memory = ConversationBufferMemory()

chain = prompt | \

 ChatOpenAI(model="gpt-4o-mini") | \

 StrOutputParser()

# Invoke the chain with a question and the memory

# will track history

question = "Who is Steve Jobs?"

response = chain.invoke(

 {"question" : question,

 "history" : memory.load\_memory\_variables({})["history"]})

print(response)

memory.save\_context(

 {"question": question},

{"answer": response} )

# Ask another question to continue the conversation

question = "What company did he found?"

response = chain.invoke({"question": question,

 "history" : memory.load\_memory\_variables({})["history"]})

print(response)

memory.save\_context({"question": question},

 {"answer": response})

Listing 2. Summarizing the history using Hugging Face transformers’ pipeline object

from transformers import pipeline

def summarize\_history():

 long\_history = memory.load\_memory\_variables({})["history"]

 # load the model to perform summarization

 summarizer = pipeline("summarization",

 model="facebook/bart-large-cnn")

 summary = summarizer(long\_history,

 max\_length=150,

 min\_length=30,

 do\_sample=False)

 # clear the memory after summarizing

 memory.clear()

 # Save summarized context

 memory.save\_context(

 { "summary": summary[0]['summary\_text'] },

 { "answer": "" })

Listing 3. Modifying the code to summarize the chat history if it has more than four history entries

while True:

 question = input('Question: ')

 if question.lower() == 'quit': break

 # Invoke the chain with a question and the

 # memory will track history

 response = chain.invoke(

 {"question" : question,

 "history" :

 memory.load\_memory\_variables({})["history"]})

 print(response)

 memory.save\_context({"question": question},

 {"answer": response})

 # if more then 4 messages, summarize the history

 if len(memory.chat\_memory.messages) > 8:

 summarize\_history()

Listing 4. Asking multiple questions at once

from langchain import PromptTemplate

from langchain\_openai import ChatOpenAI

from langchain\_core.output\_parsers \

import StrOutputParser

template = '''

Question: {question}

Answer: '''

# create the three components

prompt = PromptTemplate(

 template = template,

 input\_variables = ['question'])

model = ChatOpenAI(model="gpt-4o-mini")

output\_parser = StrOutputParser()

# create the chain

chain = prompt | model | output\_parser

# set fof questions to ask

qs = [

 {'question': 'What is the population of Singapore?'},

 {'question': 'Which comes first? Egg or Chicken?'},

]

# ask multiple qns

res = chain.invoke(qs)

print(res)

Listing 5. Modifying the prompt for language translation task

from langchain\_core.output\_parsers \

import StrOutputParser

from langchain\_openai import ChatOpenAI

from langchain import PromptTemplate

template = '''

Translate the following sentence

from {source\_language} to

{target\_language}:{sentence}

Translation:

'''

prompt = PromptTemplate(

 template = template,

 input\_variables = ['source\_language',

 'target\_language',

 'sentence']

)

chain = prompt | \

 ChatOpenAI(model="gpt-4o-mini") | \

 StrOutputParser()

chain.invoke(

 {

 'source\_language':'English',

 'target\_language':'Chinese',

 'sentence':'How are you'

 })

Listing 6. Using the Hugging Face LLM for the LangChain application

import os

from langchain import PromptTemplate

from langchain\_huggingface import HuggingFaceEndpoint

from langchain\_core.output\_parsers import StrOutputParser

# replace with your own access token

os.environ['HUGGINGFACEHUB\_API\_TOKEN'] = 'Hugging Face token'

template = '''

Question: {question}

Answer:

'''

prompt = PromptTemplate(

 template = template,

 input\_variables = ['question']

)

hub\_llm = HuggingFaceEndpoint(

 repo\_id = 'tiiuae/falcon-7b-instruct',

 # lower temperature makes the output more deterministic

 temperature = 0.1

)

chain = prompt | hub\_llm | StrOutputParser()

chain.invoke({"question": "Who is Steve Jobs?"})

Listing 7. The variable containing the block of text

text = '''

Diabetes mellitus is a chronic metabolic disorder

characterized by high blood sugar levels, which can lead

to serious health complications if not effectively

managed. There are two primary types of diabetes: Type

1 diabetes, which is an autoimmune condition where the

immune system mistakenly attacks insulin-producing beta

cells in the pancreas, leading to little or no insulin

production; and Type 2 diabetes, which is often

associated with insulin resistance and is more prevalent

in adults, though increasingly observed in children and

adolescents due to rising obesity rates. Risk factors

for developing Type 2 diabetes include genetic

predisposition, sedentary lifestyle, poor dietary

choices, and obesity, particularly visceral fat that

contributes to insulin resistance. When blood sugar

levels remain elevated over time, they can cause damage

to various organs and systems, increasing the risk of

cardiovascular diseases, neuropathy, nephropathy, and

retinopathy, among other complications. Management of

diabetes requires a multifaceted approach, which

includes regular monitoring of blood glucose levels,

adherence to a balanced diet rich in whole grains,

fruits, vegetables, and lean proteins, and engaging in

regular physical activity. In addition to lifestyle

modifications, many individuals with Type 2 diabetes

may require oral medications or insulin therapy to help

regulate their blood sugar levels. Education about the

condition is crucial, as it empowers individuals to make

informed decisions regarding their health. Furthermore,

the role of technology in diabetes management has grown significantly, with continuous glucose monitors and

insulin pumps providing real-time feedback and improving

the quality of life for many patients. As research

continues to advance, emerging therapies such as

glucagon-like peptide-1 (GLP-1) receptor agonists and

sodium-glucose cotransporter-2 (SGLT2) inhibitors are

being explored for their potential to enhance glycemic

control and reduce cardiovascular risk. Overall, with

appropriate management strategies and support,

individuals living with diabetes can lead fulfilling

lives while minimizing the risk of complications

associated with the disease.

'''

Listing 8. Performing chunking with overlap

def split\_text\_with\_overlap(text, chunk\_size, overlap\_size):

 # Split the text into sentences

 sentences = text.split('. ')

 chunks = []

 current\_chunk = ""

 for sentence in sentences:

 # Check if adding this sentence exceeds the chunk size

 if len(current\_chunk) + len(sentence) + 1 <= chunk\_size:

 if current\_chunk: # If it's not the first sentence

 current\_chunk += ". "

 current\_chunk += sentence

 else:

 # Store the current chunk

 chunks.append(current\_chunk.strip())

 # Create a new chunk with the overlap

 # Add the last `overlap\_size` sentences from

 # the current chunk

 overlap\_sentences = \

 current\_chunk.split('. ')[-overlap\_size:]

 current\_chunk = '. '.join(overlap\_sentences) + \

 ". " + sentence

 # Add any remaining chunk

 if current\_chunk:

 chunks.append(current\_chunk.strip())

 return chunks

# Define chunk size and overlap size

chunk\_size = 300

overlap\_size = 1 # Number of sentences to overlap

# Split the text into chunks with overlap

text\_chunks = split\_text\_with\_overlap(

 text, chunk\_size, overlap\_size)

# Print the resulting chunks

for i, chunk in enumerate(text\_chunks):

 print(f"Chunk {i+1}:\n{chunk}\n")

Listing 9. Chaining all the components

template = """Answer the question based only on the

following context:

{context}

Question: {question}

"""

# uses a model from OpenAI

model = ChatOpenAI(model = "gpt-4o-mini")

# creates the prompt

prompt = ChatPromptTemplate.from\_template(template)

# creats the output parser

output\_parser = StrOutputParser()

# RunnableParallel is used to run multiple processes or

# operations in parallel

setup\_and\_retrieval = RunnableParallel(

 {

 "context": retriever,

 "question": RunnablePassthrough()

 }

)

# creating the chain

chain = setup\_and\_retrieval | prompt | model | output\_parser

Listing 10. Changing the LLM to a model hosted by Hugging Face

from langchain\_core.runnables import RunnableParallel,

 RunnablePassthrough

from langchain\_core.output\_parsers import StrOutputParser

from langchain\_core.prompts import ChatPromptTemplate

from langchain.llms import HuggingFacePipeline

from transformers import pipeline

import torch

# determine the device

if torch.backends.mps.is\_available():

 device = torch.device("mps")

else:

 device = torch.device(

 "cuda" if torch.cuda.is\_available() else "cpu")

# Load a Hugging Face pipeline for text generation

generator = pipeline('text2text-generation',

 model='facebook/bart-large',

 max\_length=500,

 device=device)

# Create a LangChain LLM wrapper

model = HuggingFacePipeline(pipeline=generator)

template = """Answer the question based only on the

following context:

{context}

Question: {question}

"""

# creates the prompt

prompt = ChatPromptTemplate.from\_template(template)

# creates the output parser

output\_parser = StrOutputParser()

setup\_and\_retrieval = RunnableParallel(

 {

 "context": retriever,

 "question": RunnablePassthrough()

 }

)

# creating the chain

chain = setup\_and\_retrieval | prompt | model | output\_parser