LANGCHAIN

PROGRAMMING FOR BEGINNERS

A Step-By-Step Guide to AI Application Development With LangChain, Python, OpenAI/ChatGPT, Google/Gemini and Other LLMs

NATHAN SEBHASTIAN

TABLE OF CONTENTS

LangChain Programming For Beginners

A Step-By-Step Guide to AI Application Development With LangChain, Python, OpenAI/ChatGPT, Google/Gemini and Other LLMs

By Nathan Sebhastian

PREFACE

The goal of this book is to provide a gentle step-by-step instructions that help you learn LangChain gradually from basic to advanced.

You will see why LangChain is a great tool for building AI applications and how it simplifies the integration of language models into your projects.

We'll see how essential LangChain features such as prompt templates, chains, agents, document loaders, output parsers, and model classes are used to create a generative AI application that's smart and flexible.

After finishing this book, you will know how to use LangChain and Python to create AI-powered applications.

Working Through This Book

This book is broken down into 15 concise chapters, each focusing on a specific topic in LangChain programming.

I encourage you to write the code you see in this book and run them so that you have a sense of how LangChain development

looks like. You learn best when you code along with examples in this book.

A tip to make the most of this book: Take at least a 10-minute break after finishing a chapter, so that you can regain your energy and focus.

Also, don't despair if some concept is hard to understand. Learning anything new is hard for the first time, especially something technical like programming. The most important thing is to keep going.

Requirements

To experience the full benefit of this book, basic knowledge of Python is required.

If you need some help in learning Python, you can get my book at https://g.codewithnathan.com/beginning-python

Source Code

You can download the source code from GitHub at the following link:

https://github.com/nathansebhastian/langchain-python

Click on the 'Code' button, then click on the 'Download ZIP' link as shown below:

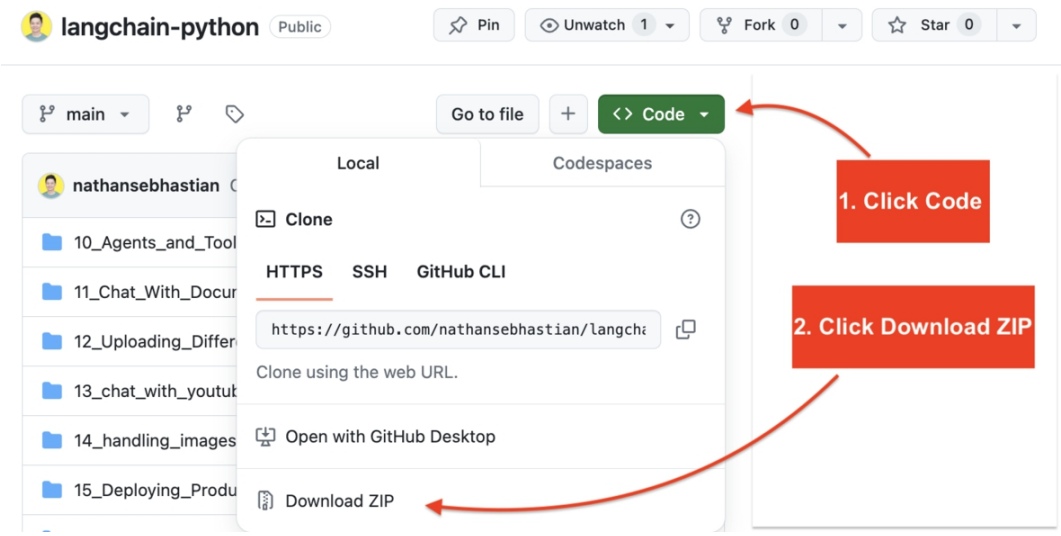

Figure 1. Download the Source Code at GitHub

You need to extract the archive to access the code. Usually, you can just double-click the archive to extract the content.

The number in the folder name indicates the chapter number in this book.

Contact

If you need help, you can contact me at nathan@codewithnathan.com.

You can also connect or follow me on LinkedIn at https://linkedin.com/in/nathansebhastian

CHAPTER 1: INTRODUCTION TO GENERATIVE AI APPLICATIONS

A Generative AI application is a computer application that can generate contextually relevant output based on a given input (or prompt).

Generative AI application came to the attention of the general public in 2022, when OpenAI released ChatGPT and quickly gained 1 million users in just 5 days:

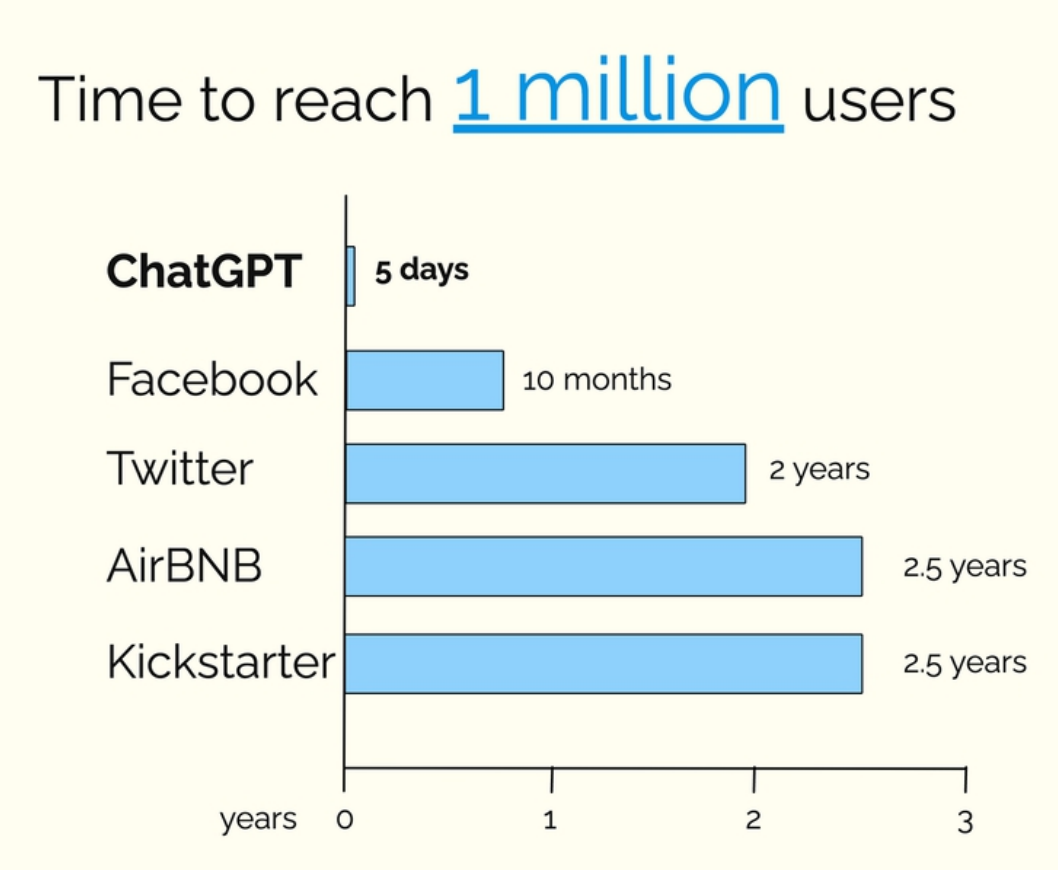

Figure 2. ChatGPT Reached 1 Million Users in 5 Days

Another example of a generative AI application is chatpdf.com, which enables users to upload a PDF and perform various tasks, such as extracting insights from the PDF.

The answers provided by chatpdf.com contain references to their source in the original PDF document, so no more flipping pages to find the source.

Behind the scenes, these generative AI applications use the power of Large Language Models to generate the answers.

What is a Large Language Model?

A Large Language Model (LLM for short) is a machine learning model that can understand and generate an output that humans can understand.

LLMs are usually trained on a vast amount of text data available on the internet so that they can perform a wide range of language-related tasks such as translation, summarization, question answering, and creative writing.

Examples of LLMs include GPT-4 by OpenAI, Gemini by Google, Llama by Meta, and Mistral by Mistral.

Some LLMs are closed-source, like GPT and Gemini, while some are open-source such as Llama and Mistral.

What is LangChain?

LangChain is an open-source framework designed to simplify the process of developing a LLM-powered application.

LangChain enables you to integrate and call LLM which powers generative AI applications by simply calling the class that represents the model.

Under the hood, LangChain will perform the steps required to interact with the language model API and manage the processing of input and output so that you can access different LLMs with minimal code change.

What's more, you can also use external data sources such as a PDF, a Wikipedia article, or a search engine result with LangChain to produce a contextually relevant response.

By using LangChain, you can develop specialized generative AI applications that are optimized for certain use cases, such as summarizing a YouTube video, extracting insights from a PDF, or writing an essay.

The Architecture of a Generative AI Application

A traditional application commonly uses the client-server architecture as follows:

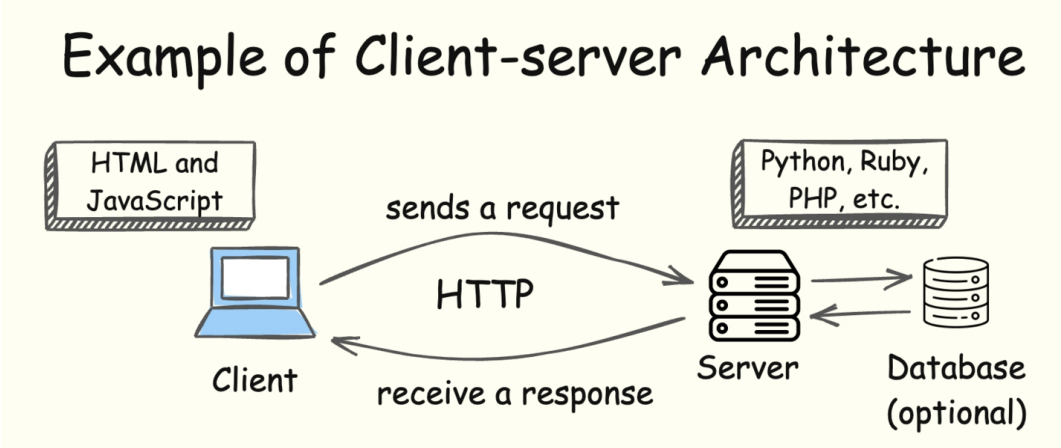

Figure 3. Client-Server Architecture

The client and server communicate using HTTP requests. When needed, a server might interact with the database to fulfill the request sent by the client.

On the other hand, a Generative AI application utilizes the power of LLM to understand human language prompts and generate relevant outputs:

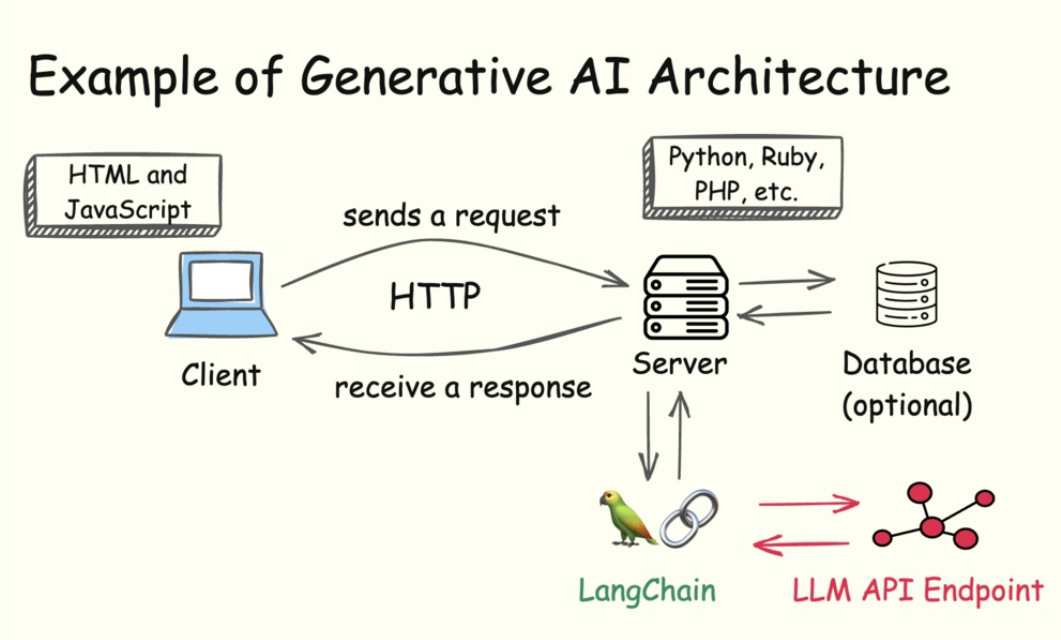

Figure 4. AI-Powered Application Architecture

While the architecture is similar to a traditional application, there's an added layer to connect to LLMs.

This added layer is where LangChain comes in, as it performs and manages tasks related to LLM, such as processing our input into a prompt that LLMs can understand. It also processes the response from LLM into a format that's accepted in traditional applications.

You'll understand more as you practice building generative AI applications in the following chapters.

For now, just think of LangChain as a management layer between your application server and the LLM.

Development Environment Set Up

To start developing AI applications with LangChain, you need to have three things on your computer:

1. A web browser
2. A code editor
3. The Python programming language

Let's install them in the next section.

Installing Chrome Browser

Any web browser can be used to browse the Internet, but for development purposes, you need to have a browser with sufficient development tools.

The Chrome browser developed by Google is a great browser for web development, and if you don't have the browser installed, you can download it here:

https://google.com/chrome

The browser is available for all major operating systems. Once the download is complete, follow the installation steps presented by the installer to have the browser on your computer.

Next, we need to install a code editor. There are several free code editors available on the Internet, such as Sublime Text, Visual Studio Code, and Notepad++.

Out of these editors, my favorite is Visual Studio Code because it's fast and easy to use.

Installing Visual Studio Code

Visual Studio Code or VSCode for short is a code editor application created for the purpose of writing code. Aside from being free, VSCode is fast and available on all major operating systems.

You can download Visual Studio Code here:

https://code.visualstudio.com

When you open the link above, there should be a button showing the version compatible with your operating system as shown below:

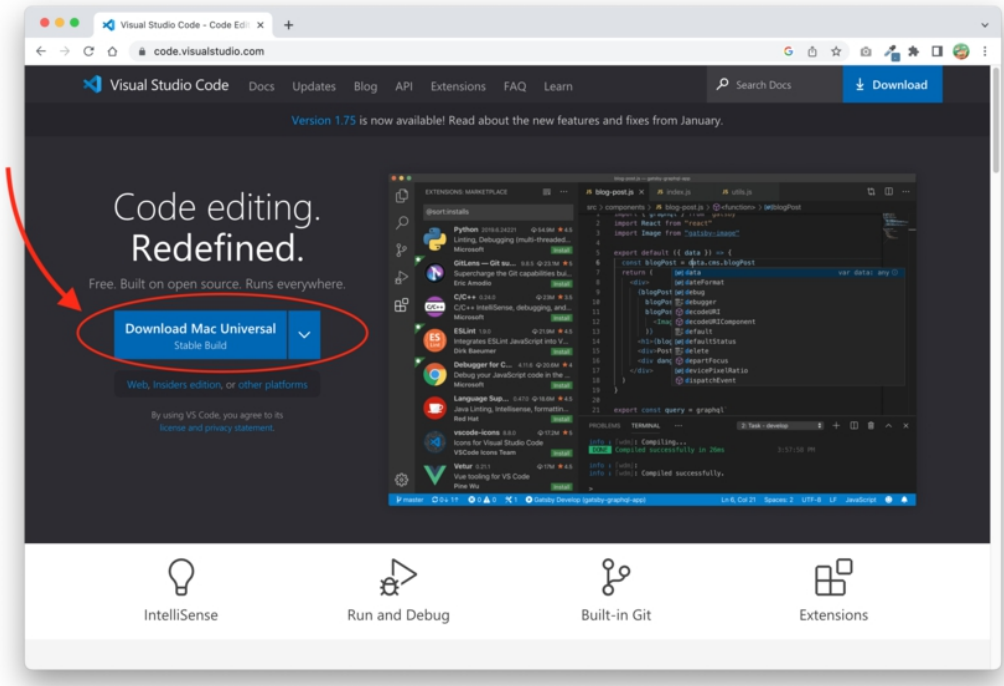

Figure 5. Downloading VSCode

Click the button to download VSCode, and install it on your computer.

Now that you have a code editor installed, the next step is to install Node.js

Installing Python

The following section will show you how to install Python on macOS, Windows, and Linux. Feel free to skip to the operating system you used.

Installing Python on macOS

If you're using the latest version of macOS, then Python should be included on your system already.

To check if you have Python installed, open the Terminal application and run the following command:

```
python3 --version
```

You should see the output as follows:

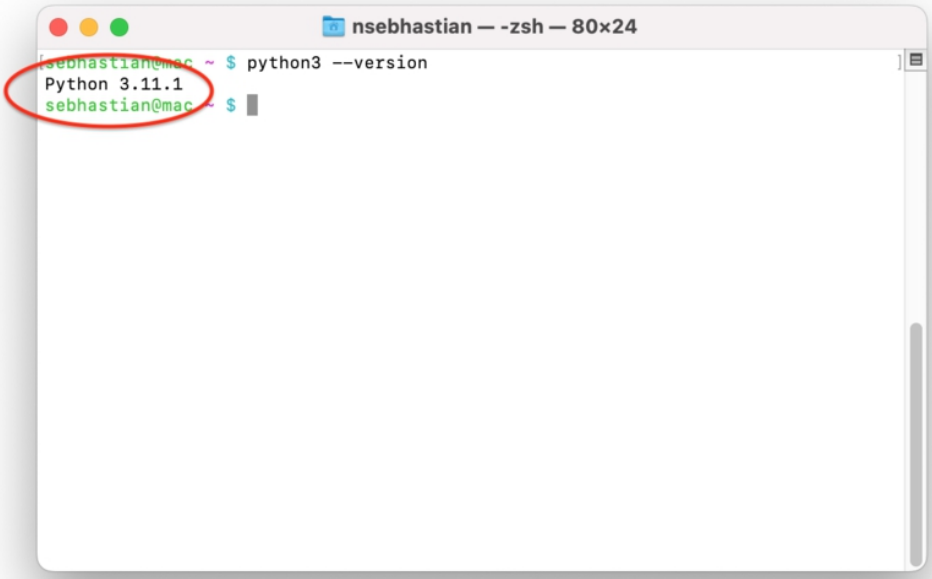

Figure 6. Checking Python Version on Mac

If you see the `command not found: python3` message, this means you need to install Python by yourself.

Open your browser and head to https://www.python.org/downloads to download the Python Interpreter for Mac.

The website should show a link to download the `.dmg` or `.pkg` installer for your Mac.

Once downloaded, just follow the installation steps shown on the screen.

After you finish installing, run the `python3 --version` command again. This time, you should be able to see the Python version in the output.

Installing Python on Windows

To install Python on Windows, you need to download the Python installer at https://www.python.org/downloads.

The website should detect the Operating System you used and offer a compatible Python installer as shown below:

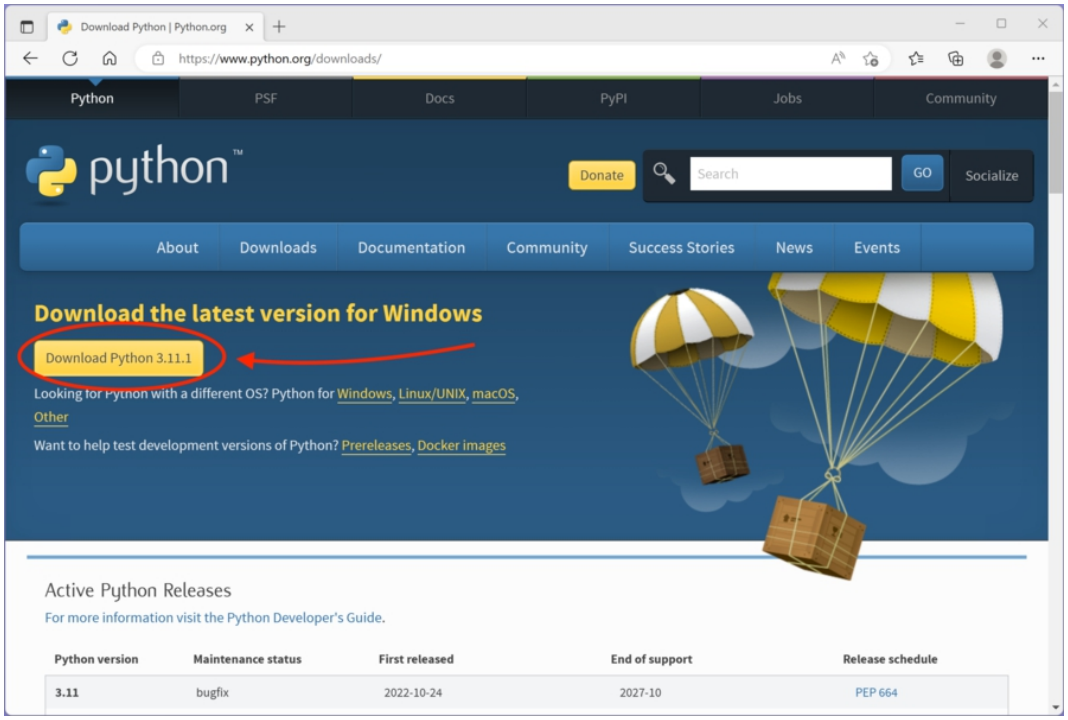

Figure 7. Installing Python on Windows

Download and run the `.exe` installer to get the latest stable version of Python on Windows.

In the setup wizard, you need to select the option **Add Python.exe to PATH** as follows:

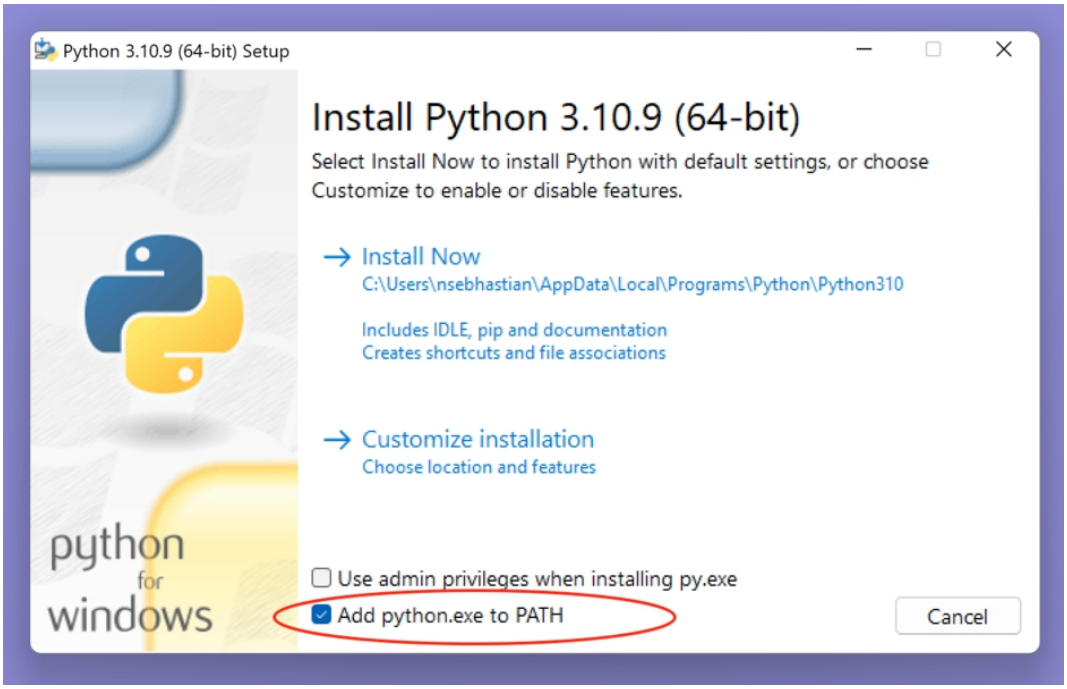

Figure 8. Adding Python to Path on Windows

This is required so that you can access Python from the Command Prompt later. Next, Select the **Install Now** option to install with the default settings.

Once done, open the Command Prompt and run the following command:

```
python --version
```

You should see a Python version number in the output like this:

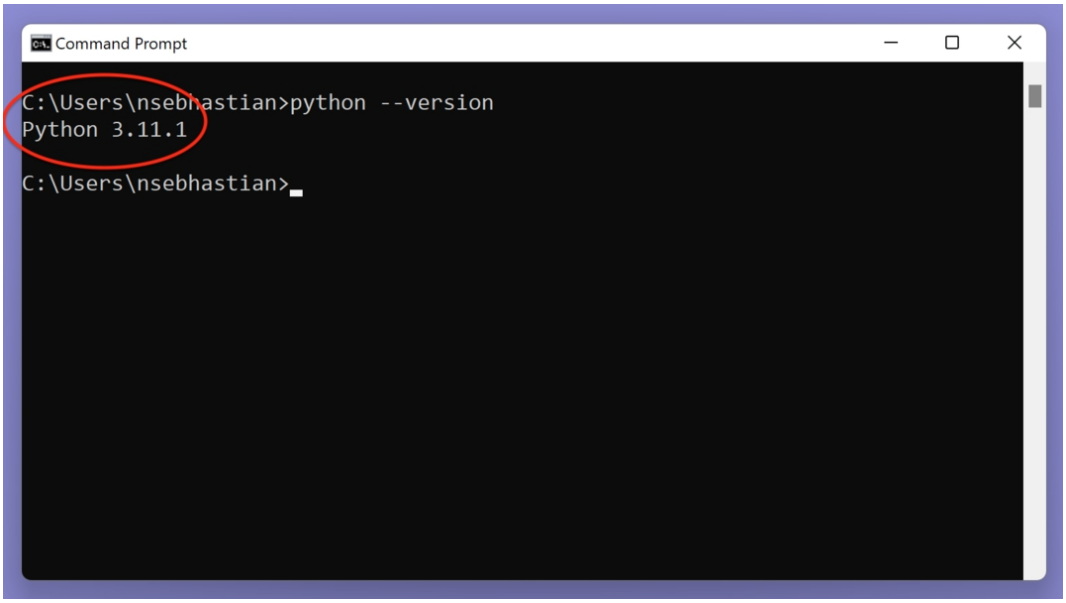

Figure 9. Windows Python Version

If that doesn't work, try running `py --version` as the Windows system commonly gives the `py` alias for the Python program.

Now you're ready to run Python code in your Windows system.

Installing Python on Linux

Linux systems are designed for software developers, so many Linux distributions already include Python by default.

If you're using the latest version of Ubuntu, try to run Python from the Terminal. Type `python3 --version` and run the command.

You should see the following response:

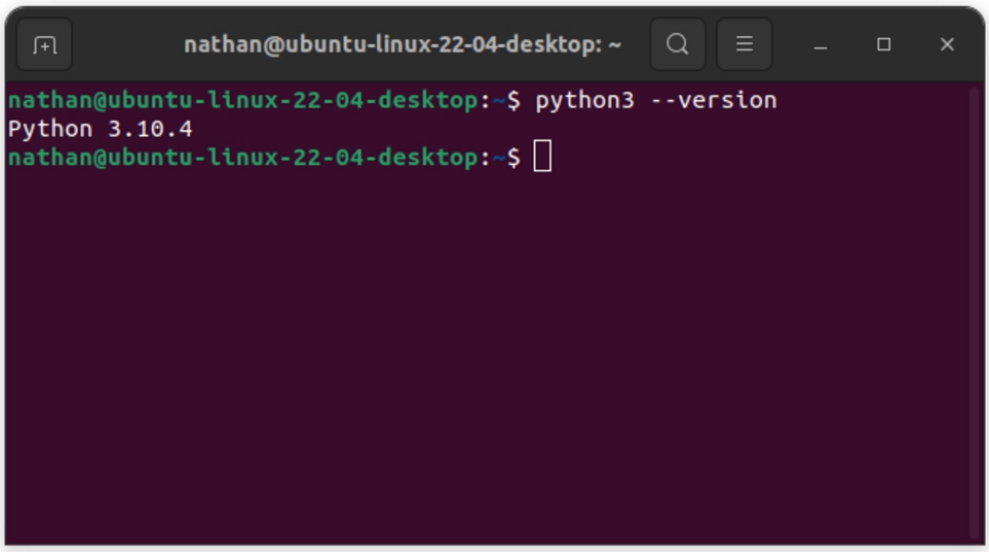

Figure 10. Ubuntu Python Version

This means that the Python interpreter is already installed on your system.

If you see Linux responds with `Command 'python3' not found`, then you can install it using `apt-get` as follows:

```
sudo apt-get install python3
```

With that, you're now able to run Python code on a Linux system.

Why the Different python and python3 Commands?

In macOS and Linux, a Python Interpreter is usually included in the OS because many applications in these two systems depend on Python.

The Python version bundled with the OS is usually Python 2, and you can verify this by running `python --version` command in Mac and Linux.

To let you install Python 3 without colliding with the bundled version, the Python installer for Mac and Linux uses the name `python3` as an alias.

This is why there are two commands that can be used to run Python: `python` and `python3`.

Because Windows didn't include a Python Interpreter, it usually has only the `python` command, while macOS and Linux can have `python` and `python3` available.

So depending on the OS you have, you need to use either `python` or `python3` command to run the Python 3 Interpreter.

Summary

In this chapter, you've learned the architecture of a generative AI application, and how LangChain takes the role of an integration layer between the server and the LLM API endpoint.

You've also installed the tools required to write and run a LangChain application on your computer.

If you encounter any issues, you can email me at nathan@codewithnathan.com and I will do my best to help you.

CHAPTER 2: YOUR FIRST LANGCHAIN APPLICATION

It’s time to create our first LangChain application.

We will create a simple question and answer application where we can ask any kind of question to a Large Language Model

First, you need to install the LangChain module using pip.

pip is a Python program used to install and manage Python packages. It’s already included when you install Python on your computer.

Open the command line, then run the command below to check pip version:

```
pip --version
```

Python packages are Python libraries and frameworks that you can use for free in your project. We’re going to use pip to install the LangChain package.

If you don’t have pip, you need to install Python using the instructions in the previous chapter.

On your command line, run the command below:

```
pip install langchain==0.2.1 python-decouple==3.8 langchain-google-genai==1.0.5
```

The `pip install` command is used to install the package you specify next to it.

To avoid breaking changes, you can specify the exact version of the package to install with `==version` as shown above.

The `langchain` package contains everything you need to start using LangChain. The `python-decouple` package is used for loading and using environment variables in your project.

The `langchain-google-genai` package contains the integration between LangChain and Google Generative AI models.

Once the installation is finished, create a folder on your computer that will be used to store all files related to this project. You can name the folder 'beginning_langchain'.

Next, open the Visual Studio Code, and select *File* > *Open Folder...* from the menu bar. Select the folder you've just created earlier.

VSCode will load the folder and display the content in the Explorer sidebar, it should be empty as we haven't created any files yet.

To create a file, right-click anywhere inside the VSCode window and select *New Text File* or *New File...* from the menu.

Once the file is created, press `Control + S` or `Command + S` to save the file. Name that file as `app.py`.

The next step is to write the code for the question and answer application.

First, import the packages required by the application as follows:

```
from langchain_google_genai import ChatGoogleGenerativeAI
from decouple import config
```

To interact with LLMs in LangChain, you need to create an object that represents the API for that LLM.

Because we want to interact with Google's LLM, we need to create an object from the `ChatGoogleGenerativeAI` class as follows:

```
GOOGLE_GEMINI_KEY = config("GOOGLE_GEMINI_KEY")

llm = ChatGoogleGenerativeAI(model="gemini-pro",
google_api_key=GOOGLE_GEMINI_KEY)
```

The `GOOGLE_GEMINI_KEY` contains the API key which you're going to get in the next section.

For now, you just need to understand that the `ChatGoogleGenerativeAI` object represents the Google LLM.

When instantiating the `llm` object, you can choose the `model` you want to interact with, then pass the API key under the `google_api_key` parameter.

Next, write the code for a simple question and answer application as follows:

```
print("Q & A With AI")
print("=============")

question = "What's the currency of Thailand?"
print("Question: " + question)

response = llm.invoke(question)

print("Answer: " + response.content)
```

Here, we simply print some text showing the question we want to ask the model.

The `llm.invoke()` method will send the input `question` to the LLM and return an answer.

The answer is stored under the `content` attribute, so we print `response.content` in the code above.

While the application is ready, we still need to get the Google Gemini API key used in this application.

Getting Google Gemini API Key

To get the API key, you need to visit the Gemini API page at https://ai.google.dev/gemini-api

On the page, you need to click the 'Get API Key in Google AI Studio' button as shown below:

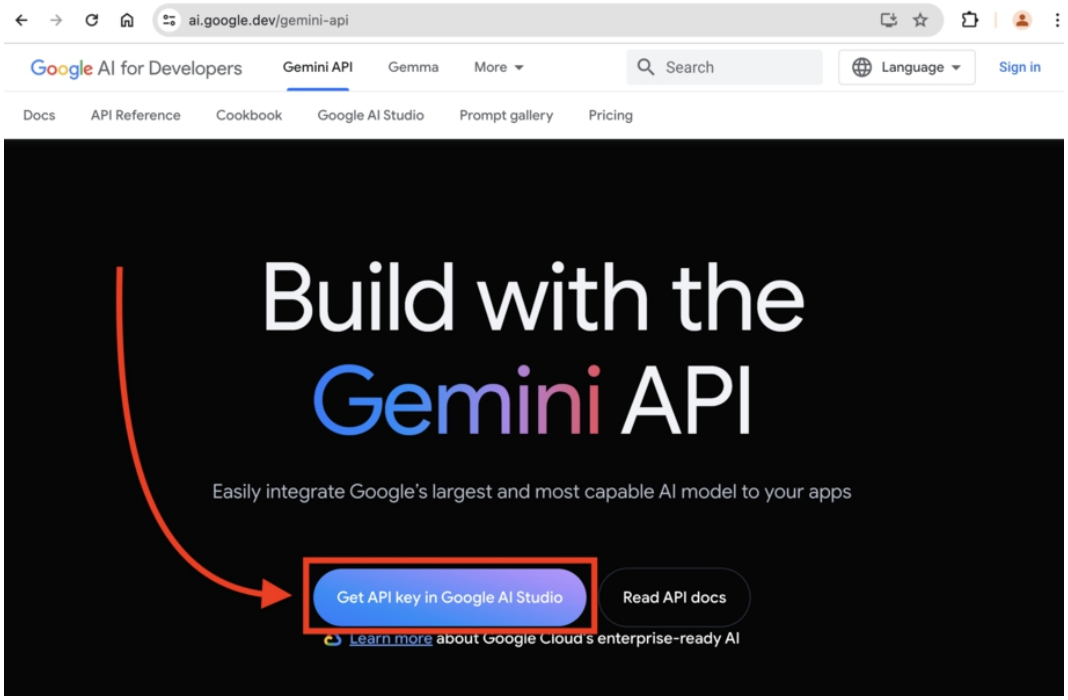

Figure 11. Get Gemini API Key

From there, you'll be taken to Google AI Studio.

Note that, you might be shown the page below when clicking the button:

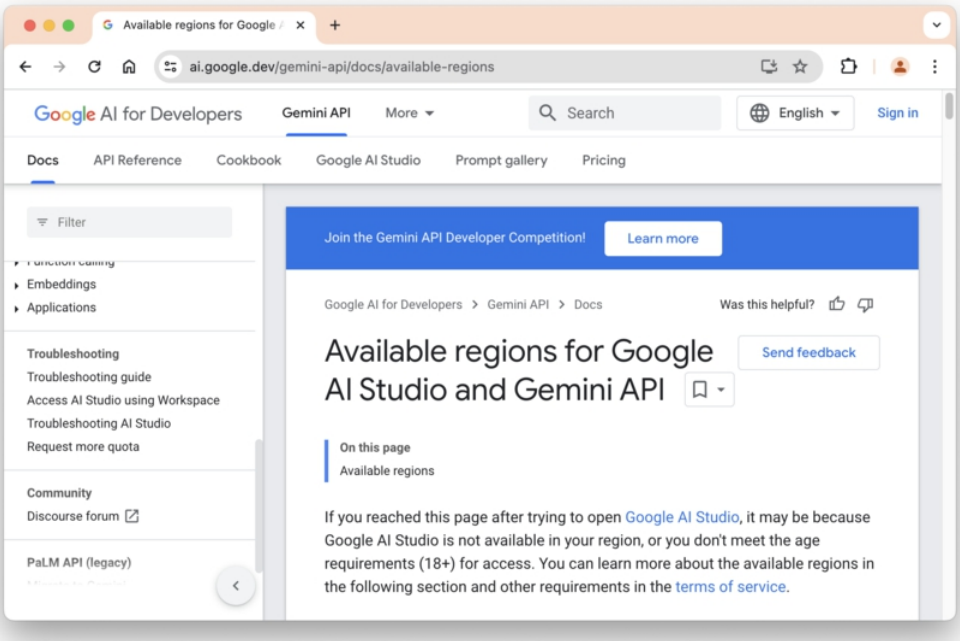

Figure 12. Google AI Studio Available Regions Page

This page usually appears when you are located in a region that's not served by Google AI Studio.

One way to handle this is to use a VPN service, but I would recommend you use another LLM instead, such as OpenAI or Ollama which I will show in the next chapters.

If this is your first time accessing the studio, it will show you the terms of service like this:

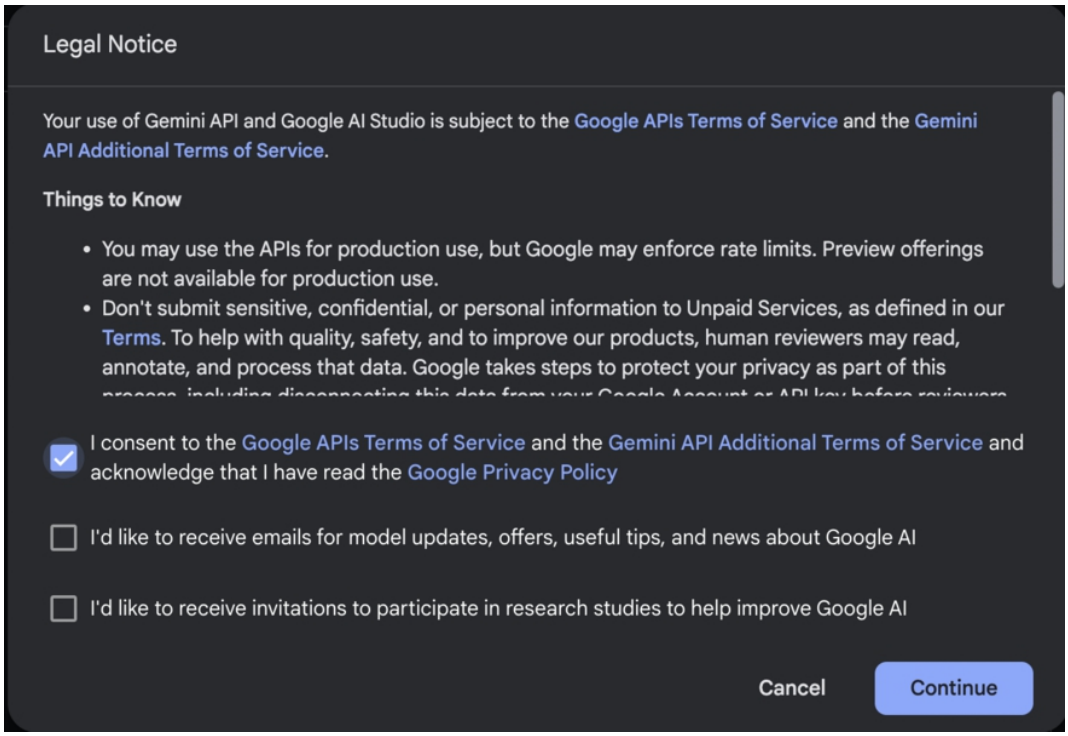

Figure 13. Google AI Studio Terms of Service

Just check on the 'I consent' option, then click 'Continue'.

Now click the 'Create API Key' button to create the key:

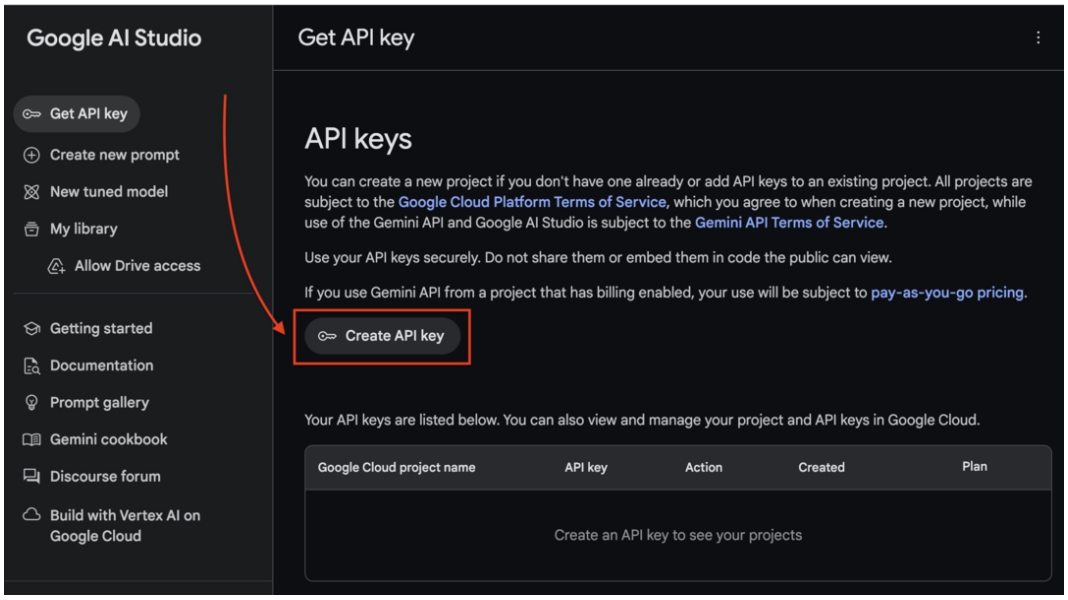

Figure 14. Gemini Create API Key

If you're asked where to create the API Key, select create in new project:

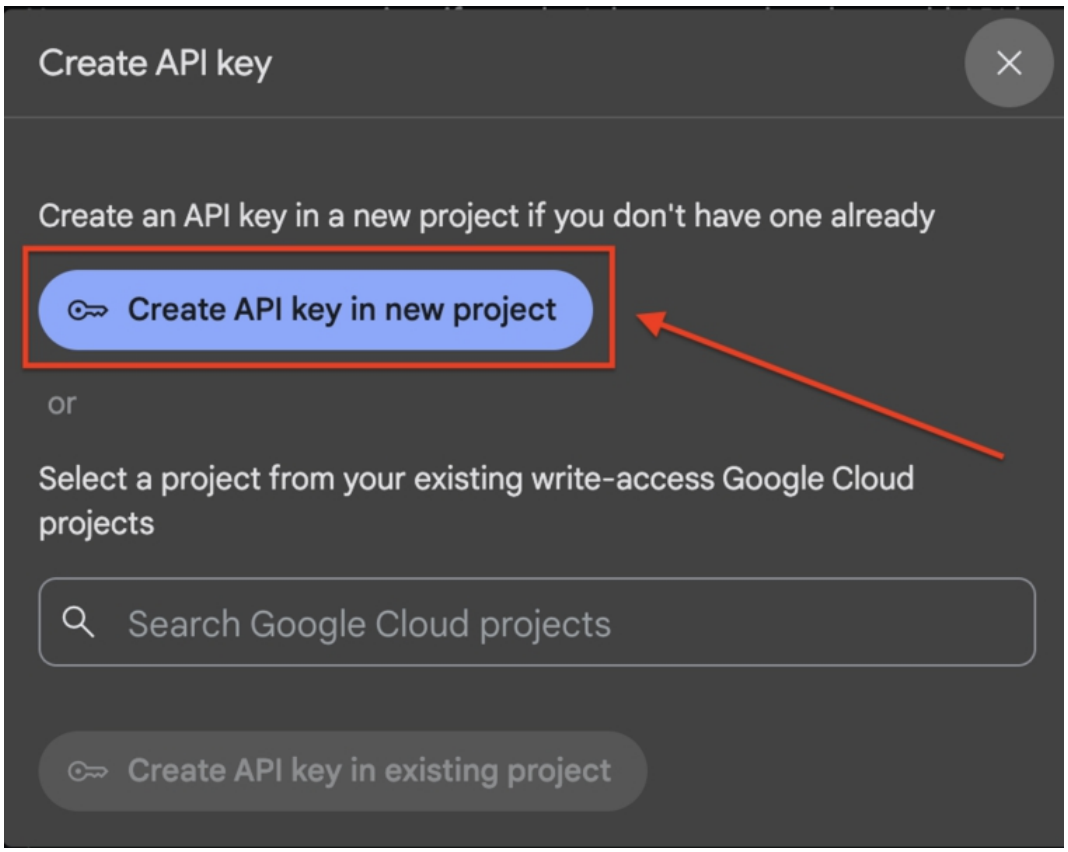

Figure 15. Create API Key in New Project

Google will create a Cloud project and generate the key for you.

After a while, you should see the key shown in a pop-up box as follows:

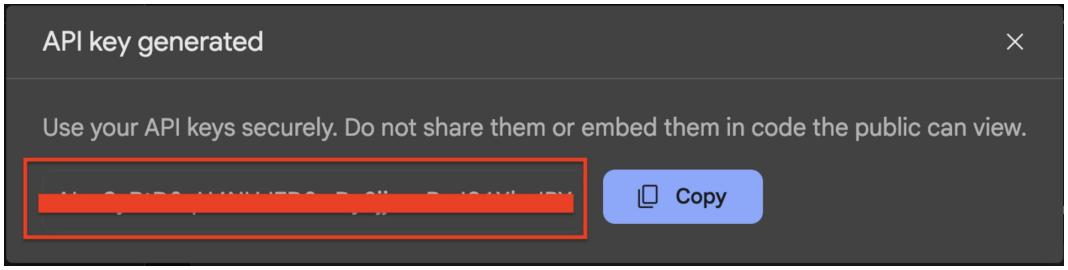

Figure 16. Gemini API Key Generated

Copy the API key string, then create a `.env` file in your application folder with the following content:

```
GOOGLE_GEMINI_KEY='Your Key Here'
```

Replace the `Your Key Here` string above with your actual API key.

Running the Application

With the API key set, you are ready to run the LangChain application.

On the terminal, run `.py` file using Python as follows:

```
python app.py
```

You should see the following output in your terminal:

```
Q & A With AI
=============
Question: What's the currency of Thailand?
Answer: Thai baht
```

This means you have successfully created your first LangChain application and interacted with Google's Gemini LLM using the API key.

Each LLM model has its own characteristics. The 'gemini-pro' model usually answers a question directly with no extra information.

You can try changing the model to 'gemini-1.5-flash-latest' as shown below:

```
llm = ChatGoogleGenerativeAI(model="gemini-1.5-flash-latest",
google_api_key=GOOGLE_GEMINI_KEY)
```

Now run the `.py` file again using Python, and the answer is a bit different this time:

```
Q & A With AI
=============
Question: What's the currency of Thailand?
Answer: The currency of Thailand is the **Thai baht**, which is abbreviated as
**THB** or **฿**.
```

Here, the 'gemini-1.5-flash' repeats the question first, then gives more information such as the currency abbreviation and symbol.

The asterisk ** symbols around THB and ฿ are meant to make the text appear in bold, but it's rendered as-is in the terminal.

Now try replacing the `question` variable with any question you want to ask the LLM.

Resource Exhausted Error

When using Google Gemini, you might see an error like this when running the application:

```
ResourceExhausted: 429 Resource has been exhausted (e.g. check quota)..
```

This error occurs because the free tier resource has been exhausted. You need to try again at a later time.

Summary

The code for this chapter is available in the *02_Simple_Q&A_Gemini* folder.

In this chapter, you've created and run your first LangChain application. Congratulations!

The application can connect to Google's Gemini LLM to ask questions and get answers.

In the next chapter, we're going to learn how to use OpenAI's GPT model in LangChain.

CHAPTER 3: USING OPENAI LLM IN LANGCHAIN

In the previous chapter, you've seen how to communicate with Google's Gemini model using LangChain.

In this chapter, I will show you how to use OpenAI in LangChain as an alternative.

But keep in mind that the OpenAI API has no free tier. It used to give away $5 worth of API usage, but it seems to have been quietly stopped.

So if you want to use OpenAI API, you need to buy the minimum amount of credit, which is $5 USD.

Getting Started With OpenAI API

OpenAI is an AI research company that aims to develop and promote capable AI software for the benefit of humanity. The famous ChatGPT is one of the products developed by OpenAI.

Besides the ChatGPT application, OpenAI also offers GPT models, the LLM that powers ChatGPT, in the form of HTTP API endpoints.

To use OpenAI's API, you need to register an account on their website at https://platform.openai.com.

After you sign up, you can go to https://platform.openai.com/api-keys to create a new secret key.

When you try to create a key for the first time, you'll be asked to verify by adding a phone number:

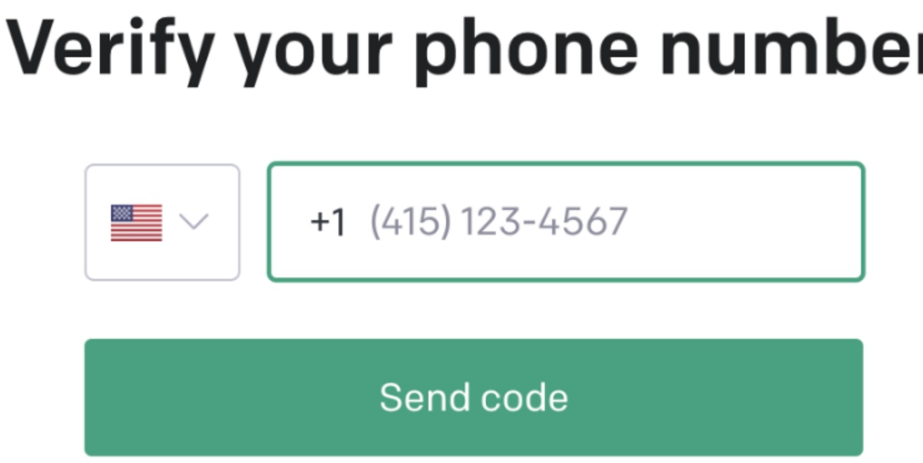

Figure 17. OpenAI Phone Verification

OpenAI only uses your phone number for verification purpose. You will receive the verification code through SMS.

Once you're verified, you're going to be asked to add credit balance for API usage. If not, go to https://platform.openai.com/account/billing to add some credits.

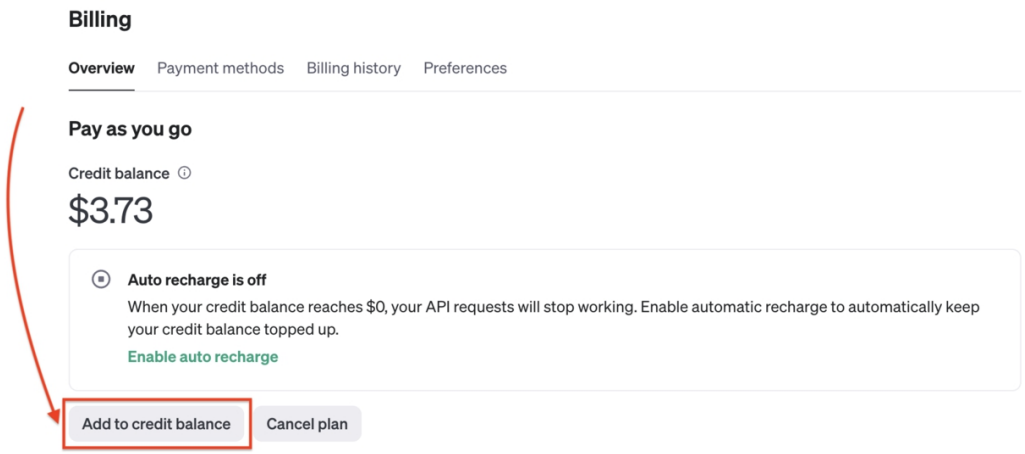

Figure 18. OpenAI Adding Credits

OpenAI receives payment using credit cards, so you need to have one. The lowest amount you can buy is $5 USD, and it will be more than enough to run all the examples in this book using OpenAI.

Alternatively, if you somehow get the $5 free trial credits, then you don't need to set up your billing information.

Next, input the name and select the project the key will belong to:

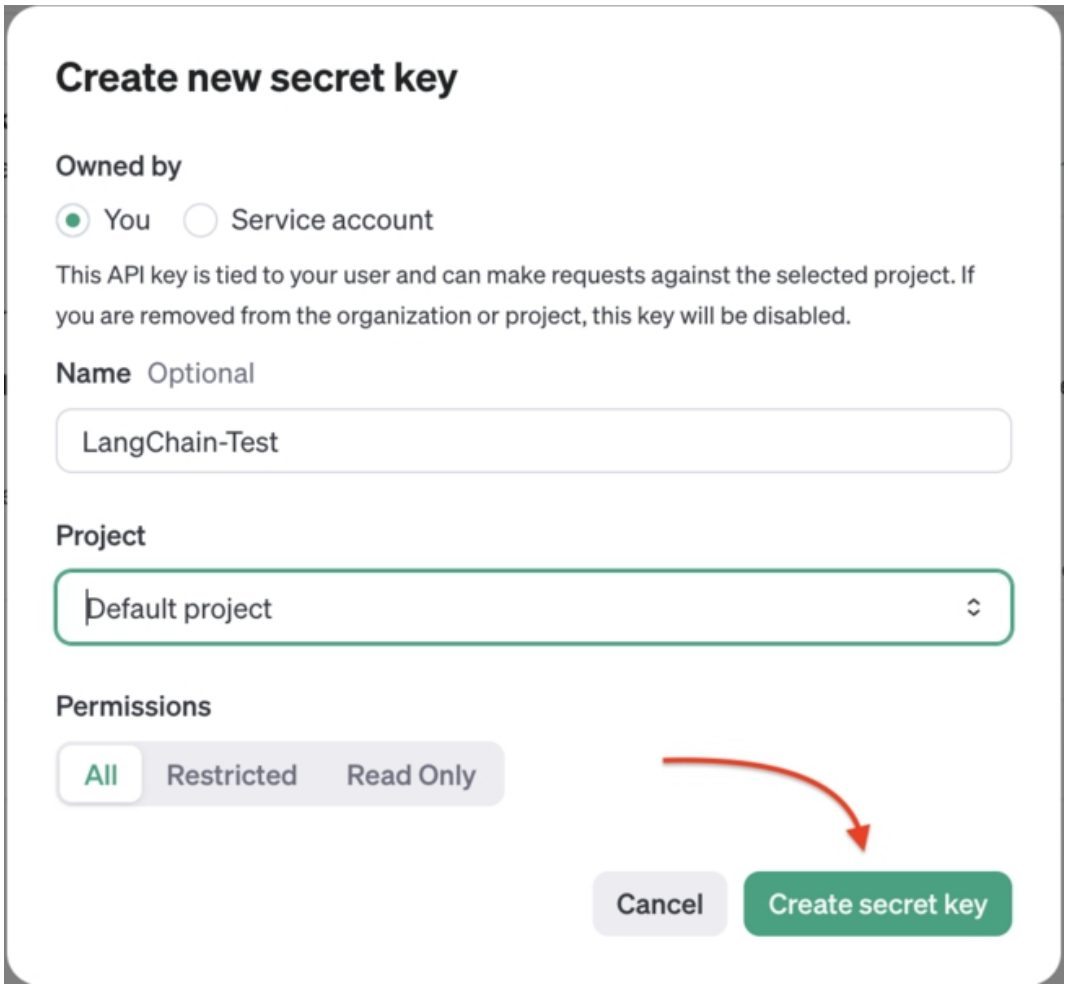

Figure 19. OpenAI Create API Key

Click the 'Create secret key' button, and OpenAI will show you the generated key:

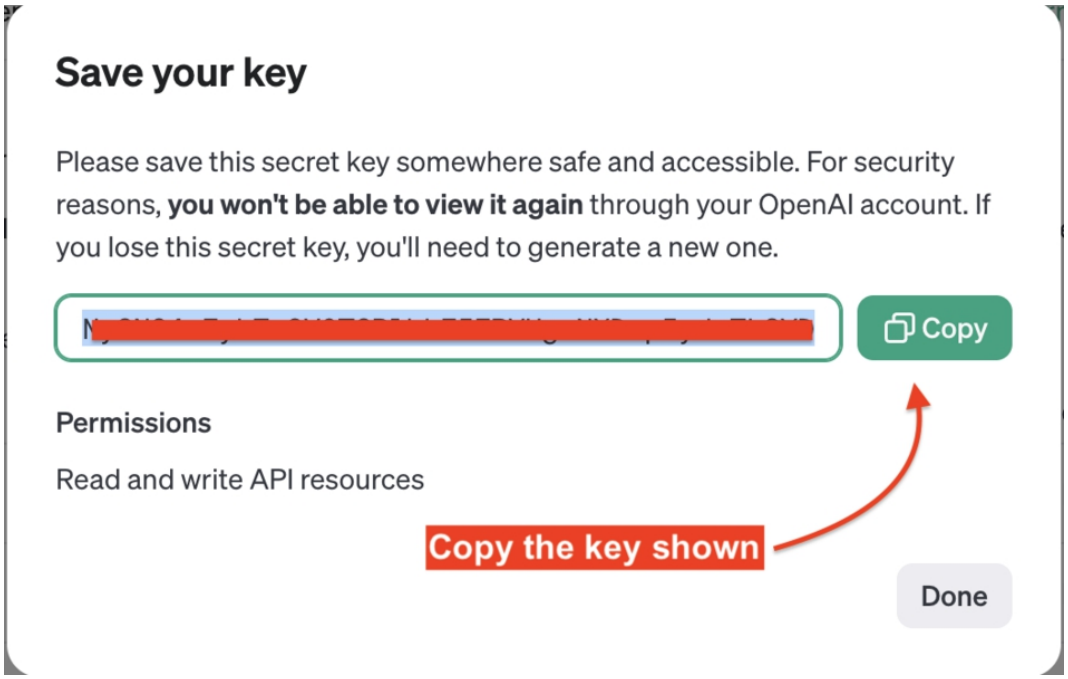

Figure 20. OpenAI Copy API Key

You need to copy and paste this API key into the `.env` file of your project:

```
OPENAI_KEY='Your Key Here'
```

Now that you have the OpenAI key ready, it's time to use it in the LangChain application.

Integrating OpenAI With LangChain

To use OpenAI in LangChain, you need to install the `langchain_openai` package using pip:

```
pip install langchain-openai==0.1.7
```

Once the package is installed, create a new file named `app_gpt.py` and import the `ChatOpenAI` class from the package.

We also need to add the `OPENAI_KEY` using `config()` as shown below:

```
from decouple import config
from langchain_openai import ChatOpenAI

OPENAI_KEY = config("OPENAI_KEY")

llm = ChatOpenAI(model="gpt-4o", api_key=OPENAI_KEY)
```

You can change the `model` parameter with the model you want to use. As of this writing, GPT-4o is the latest LLM released by OpenAI.

Let's ask GPT the same question we asked to Gemini:

```
print("Q & A With AI")
print("=============")

question = "What's the currency of Thailand?"
print("Question: " + question)

response = llm.invoke(question)

print("Answer: " + response.content)
```

Save the changes, then run the file using Python:

```
python app_gpt.py
```

You should get a response similar to this:

```
Q & A With AI
=============
Question: What's the currency of Thailand?
Answer: The currency of Thailand is the Thai Baht. Its ISO code is THB.
```

This means the LangChain application successfully communicated with the GPT chat model from OpenAI.

Awesome!

You can try asking another question by changing the `question` variable value.

ChatGPT vs Gemini: Which One To Use?

Both ChatGPT and Gemini are very capable of performing the tasks we want them to do in this book, so it's really up to you.

In the past, OpenAI used to give $5 credits for free. But it seems no longer to be the case, as many people in the OpenAI forum said they don't get it after registering.

On the other hand, Google is offering a free tier for the Gemini model in exchange for using our data to train the model, so it's okay to use it for learning and exploring LangChain.

Still, Google has the right to stop the free tier at any time, so let me introduce you to one more way to use LLMs in LangChain.

This time, we're going to use open-source models.

Summary

The code for this chapter is available in the folder *03_Using_OpenAI* from the book source code.

In this chapter, you've learned how to create OpenAI API key and use it in a LangChain application.

Here, we start to see one of the benefits of using LangChain, which is easy integration with LLMs of any kind.

LangChain represents the LLMs as packages that you can install and import into your project.

You only need to create an instance of the model class and run the `invoke()` method to access the LLM.

Whenever you need to use another LLM, you just change the `llm` variable and pass the required API key.

CHAPTER 4: USING OPEN-SOURCE LLMS IN LANGCHAIN

The LangChain library enables you to communicate with LLMs of any kind, from proprietary LLMs such as Google's Gemini and OpenAI's GPT to open-source LLMs like Meta's Llama and Mistral.

This chapter will show you how to use open-source models in LangChain. Let's jump in.

Ollama Introduction

Ollama is a tool used to run LLMs locally. It handles downloading, managing, and opening HTTP API endpoints for the models you want to use on your computer.

To get started, head over to https://ollama.com and click the 'Download' button.

From there, you can select the version for your Operating System:

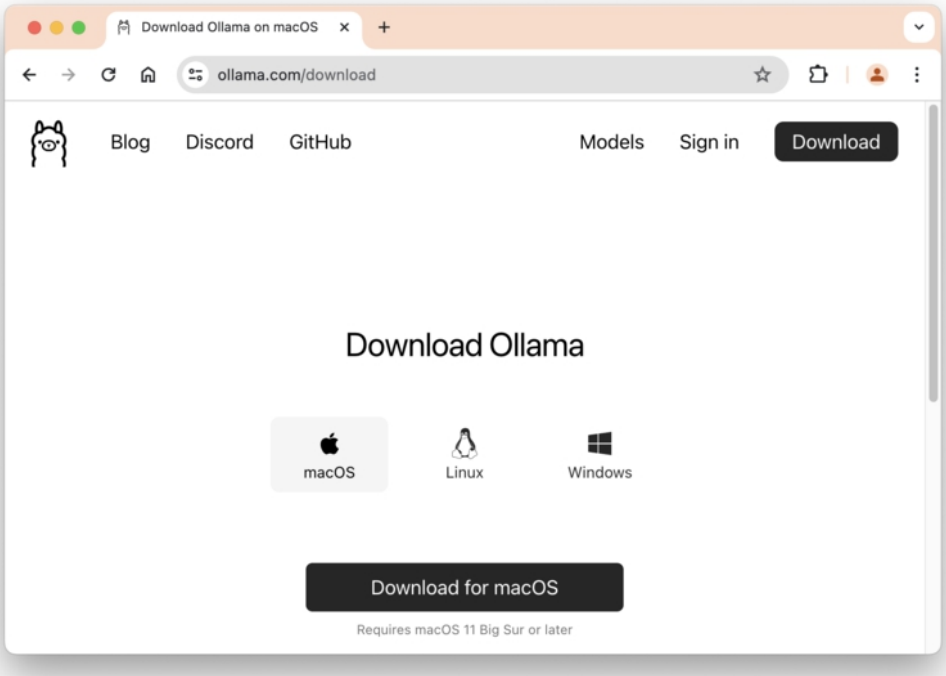

Figure 21. Downloading Ollama

Once downloaded, open the package and follow the instructions until you are asked to install the command line tool as follows:

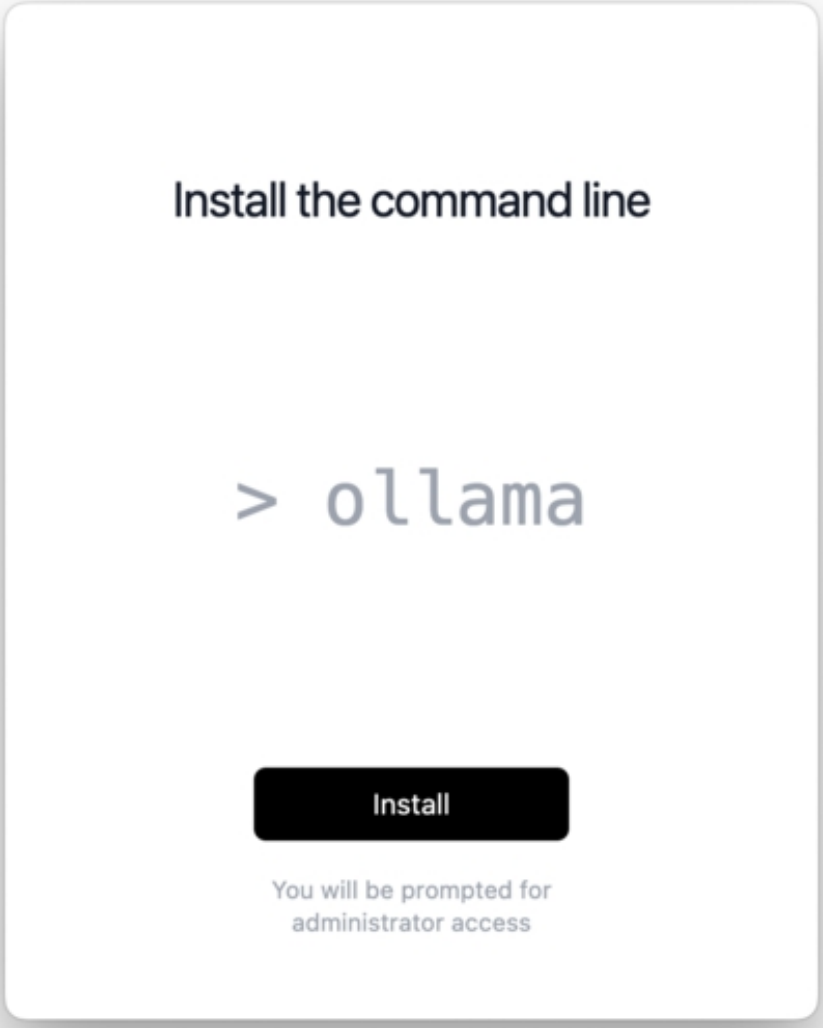

Figure 22. Installing Ollama Terminal Command

Go ahead and click the 'Install' button.

Once the installation is finished, Ollama will show you how to run a model:

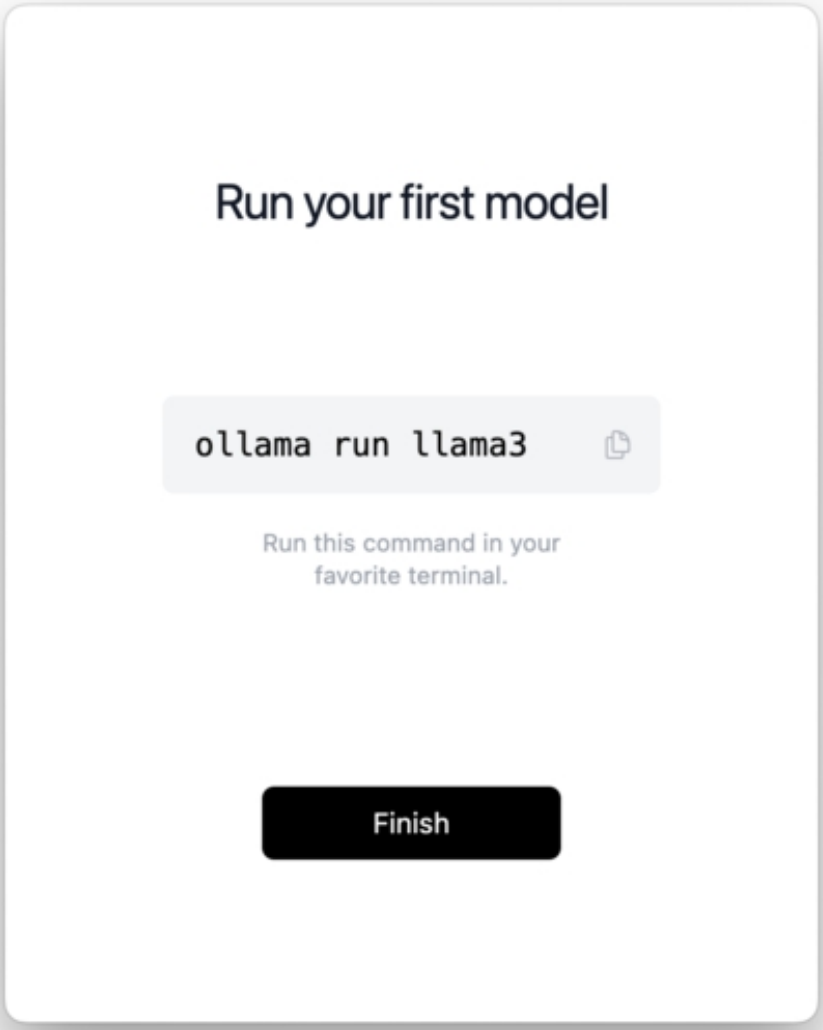

Figure 23. Ollama Running Your First Model

But since Llama 3 is an 8 billion parameter model, the model is quite large at 4.7 GB.

I recommend you run the Gemma model instead, which has 2 billion parameters:

```
ollama run gemma:2b
```

The Gemma model is a lightweight model from Google, so you can think of it as an open-source version of Google Gemini.

The Gemma 2B model is only 1.7 GB in size, so it comes in handy when you want to try out ollama.

Once the download is finished, you can immediately use the model from the terminal. Ask it a question as shown below:

Figure 24. Example of Asking Gemma in Ollama

To exit the running model, type `/bye` and press Enter.

As long as Ollama is running on your computer, the Ollama API endpoint is accessible at localhost:11434 as shown below:

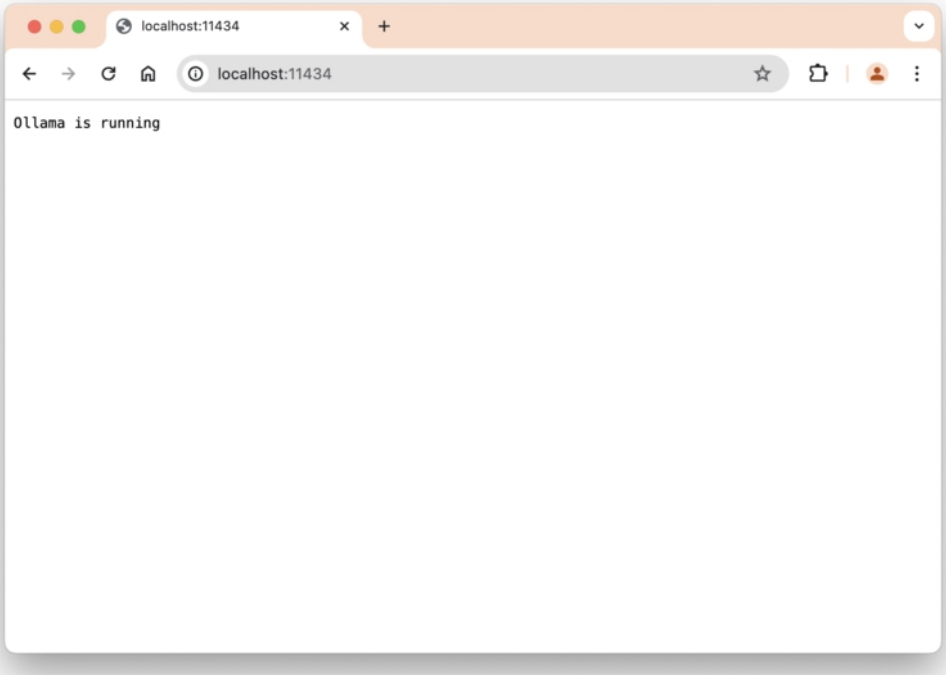

Figure 25. Ollama Localhost API Endpoint

LangChain will use this API endpoint to communicate with Ollama models, which we're going to do next.

Using Ollama in LangChain

To use the models downloaded by Ollama, you need to import the `ChatOllama` class from the `langchain_community.chat_models` module.

Create a new file named `app_ollama.py` and import the Ollama chat model as shown below:

```
from langchain_community.chat_models import ChatOllama

llm = ChatOllama(model="gemma:2b")

print("Q & A With AI")
print("=============")
```

```
question = "What's the currency of Thailand?"
print("Question: " + question)

print("Answer: " + response.content)
```

You also need to assign the `ChatOllama` object to the `llm` variable as shown above.

Because Ollama is open-source and local, you don't need to add an API key when using it.

Now you can run the file using Python to communicate with the LLM. You should have a response similar to this:

```
Q & A With AI
=============
Question: What's the currency of Thailand?
Answer: The currency of Thailand is the Thai baht (THB). It is subdivided into
100 sen. The baht is denoted by the symbol THB.
```

Note that because the LLM model is running on your computer, the answer may take longer when compared to the Gemini and GPT models.

And that's how you use Ollama in LangChain. If you want to use other open-source models, you need to download the model with Ollama first:

```
ollama pull mistral
```

The `pull` command downloads the model without running it on the command line.

After that, you can switch the `model` parameter when creating the `ChatOllama` object:

```
# Switch the model
llm = ChatOllama(model="mistral")
```

Remember that the larger the model, the longer it takes to run.

The common guideline is that you should have at least 8 GB of RAM available to run the 7B models, 16 GB to run the 13B models, and 32 GB to run the 33B models.

There are many open-source models that you can run using Ollama, such as Google's Gemma and Microsoft's Phi-3.

You can explore https://ollama.com/library to see all available models.

Again, Which One To Use?

So far, you have explored how to use Google Gemini, OpenAI GPT, and Ollama open-source models. Which one to use in your application?

I recommend you use OpenAI GPT if you can afford it because the API isn't rate-limited and the result is great.

If you can't use OpenAI GPT for any reason, then you can use Google Gemini free tier if it's available in your country.

If not, you can use Ollama and download the Gemma 2B model or Llama 3, based on your computer's RAM capacity.

Unless specifically noted, I'm going to use OpenAI GPT for all example codes shown in this book.

But don’t worry because replacing the LLM part in LangChain is very easy. You only need to change the `llm` variable itself as shown in this chapter.

You can get the code examples that use Gemini and Ollama on the repository.

Summary

The code for this chapter is available in the folder *04_Using_Ollama* from the book source code.

In this chapter, you’ve learned how to use open-source LLMs using Ollama and LangChain.

Using Ollama, you can download and run any Large Language Models that are open-source and free to use.

If you look at the Ollama website, you will find many models that are very capable and can even match proprietary models such as Gemini and ChatGPT in performance.

If you are worried about the privacy of your data and want to make sure that no one uses it for training their LLMs, then using open-source LLMs like Llama 3, Mistral, or Gemma can be a great choice.

CHAPTER 5: ADDING WEB GUI WITH STREAMLIT

So far, we have used Python to run the simple question and answer application.

Instead of using the terminal to communicate with LLMs, let's create a simple web-based user interface for interacting with LLMs.

To create the web interface, we're going to use the Streamlit module.

Streamlit Introduction

Streamlit is an open-source Python framework that you can use to build data-oriented applications with only a few lines of code.

By using Streamlit and LangChain, you can build a web-based interface for interacting with LLMs of any kind.

To start using Streamlit, you need to install the package using pip as follows:

```
pip install streamlit==1.35.0
```

Next, import the `streamlit` module to create a header and a textbox as shown below:

```
import streamlit as st

# llm...

st.title("Q & A With AI")

question = st.text_input("Your Question")

if question:
    response = llm.invoke(question)
    st.write(response.content)
```

The `st.title()` creates a <h1> element and renders the text you passed as its argument.

The `st.text_input()` creates a text input. The return value is any value you put on the box.

After that, use an `if` statement to check if the `question` variable contains a question.

When the `question` is not empty, we run the `llm.invoke()` method and write the returned response to Streamlit.

To run a Streamlit application, you need to use `streamlit` instead of `python` on the command line:

```
streamlit run app.py
```

Streamlit will open your web browser and navigate to the localhost:8501, where the Streamlit application is running:

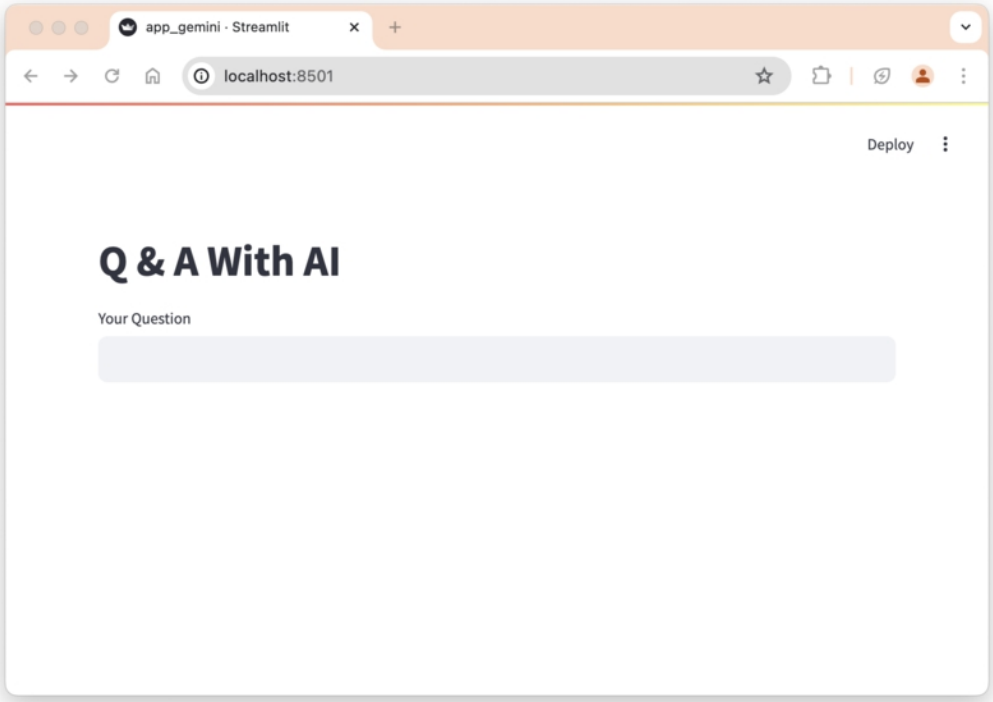

Figure 26. Opening Streamlit in Web Browser

You can type your question in the box and press Enter.

The LLM response will be written below the question box:

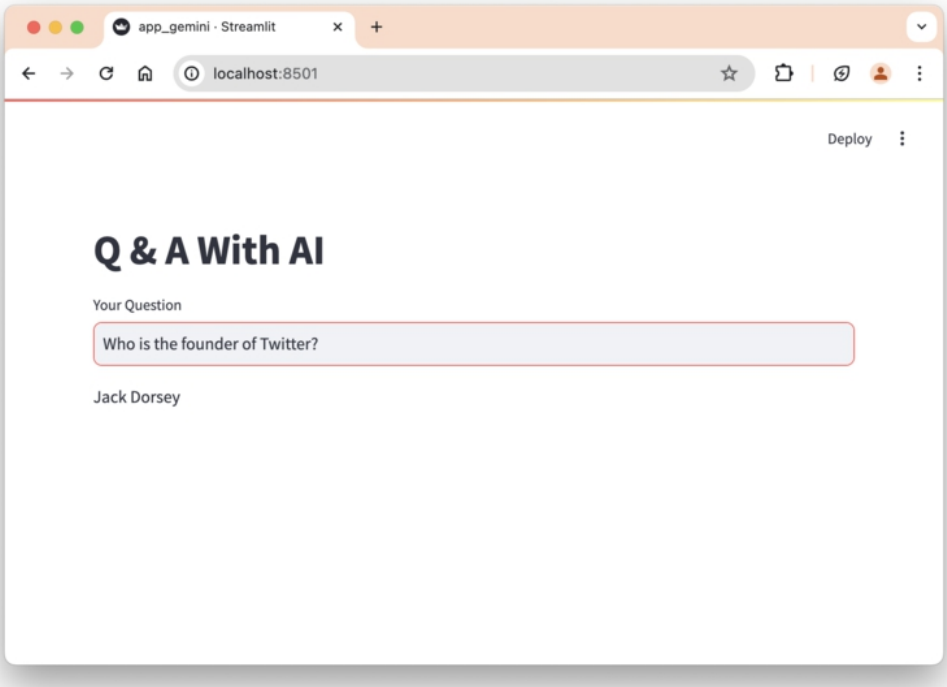

Figure 27. Streamlit Output Example

And that's how you use Streamlit to create a simple interface for LLM-powered applications.

Summary

The code for this chapter is available in the folder *05_Adding_Streamlit_GUI* from the book source code.

In this chapter, you've learned how to use Streamlit to quickly create a web-based interface for interacting with a Large Language Model through LangChain.

We're going to explore Streamlit more as we build more advanced AI applications.

In the next chapter, you’re going to learn about prompt templates. See you there!

CHAPTER 6: LANGCHAIN PROMPT TEMPLATES

The LangChain prompt template is a Python class used to create a specific prompt (or instruction) to send to a Large Language Model.

By using prompt templates, we can reproduce the same instruction while requiring minimal input from the user.

To show you an example, suppose you are creating a simple AI application that only gives currency information of a specific country.

Based on what we already know, the only way to do this is to keep repeating the question while changing the country as follows:

```
question = "What is the currency of Thailand?"
question = "What is the currency of Malaysia?"
question = "What is the currency of India?"
```

Instead of repeating the question, you can create a template for that question as follows:

```
st.title("Currency Info")

country = st.text_input("Input Country")

if country:
    question = "What is the currency of " + country + "?"
    response = llm.invoke(question)
    st.write(response.content)
```

Now you only need to give the country name on the input box to run the prompt.

You can use a plain string or f-string to create a prompt template, but LangChain recommends you use the prompt template class for effective reuse. Let me show you how.

Creating a Prompt Template

To create a prompt template, you need to import the `PromptTemplate` class from `langchain.prompts` as shown below:

```
from langchain.prompts import PromptTemplate
```

The next step is to create the prompt itself.

You can create a variable named `prompt`, then pass the call to `PromptTemplate()` constructor to that variable:

```
prompt = PromptTemplate()
```

When calling the `PromptTemplate()` constructor, you need to pass two arguments:

1. `input_variables` - A list of the names of the variables used in the template

2. `template` - A string for the prompt template itself

Here's an example of the complete `PromptTemplate()` call:

```
prompt = PromptTemplate(
    input_variables=["country"],
    template="What is the currency of {country}? Answer in one short
paragraph",
)
```

Now you can use the `prompt` object when calling the `llm.invoke()` method.

You need to call the `prompt.format()` method and pass the variable specified in the `input_variables` parameter as shown below:

```
country = st.text_input("Input Country")

if country:
    response = llm.invoke(prompt.format(country=country))
    st.write(response.content)
```

Now run the Streamlit application and ask for the currency of a specific country.

Here's an example of asking for the currency of Spain:

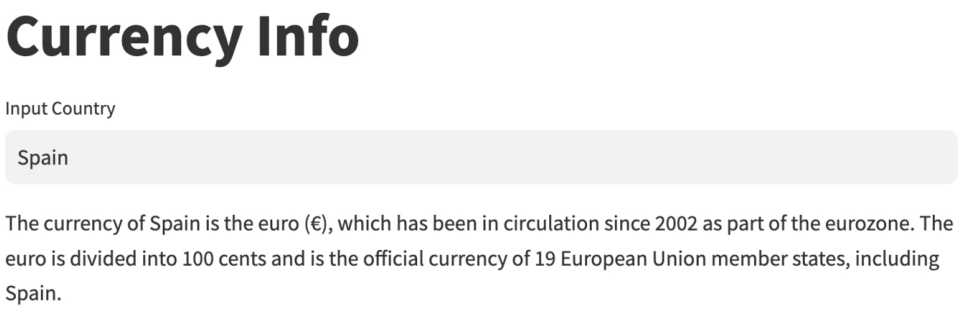

Figure 28. LLM Response

The `PromptTemplate` class provides a structure from which you can construct a specific prompt.

Prompt Template With Multiple Inputs

The prompt template can accept as many inputs as you need in your template string.

For example, suppose you want to control the amount of paragraph and the language of the answer. You can add two more variables to the prompt template like this:

```
prompt = PromptTemplate(
    input_variables=["country", "paragraph", "language"],
    template="What is the currency of {country}? Answer in {paragraph} short
paragraph in {language}",
)
```

Now below the `country` input, add two more inputs as shown below:

```
country = st.text_input("Input Country")
paragraph = st.number_input("Input Number of Paragraphs", min_value=1,
max_value=5)
language = st.text_input("Input Language")
```

Notice that the `paragraph` input is limited between 1 and 5 to avoid generating a long article.

On the `if` statement, add two more variables to check using the `and` operator, then pass the variables when calling the `prompt.format()` method:

```
if country and paragraph and language:
    response = llm.invoke(prompt.format(country=country, paragraph=paragraph,
language=language))
    st.write(response.content)
```

And that's it. Now you can try running the web application as shown below:

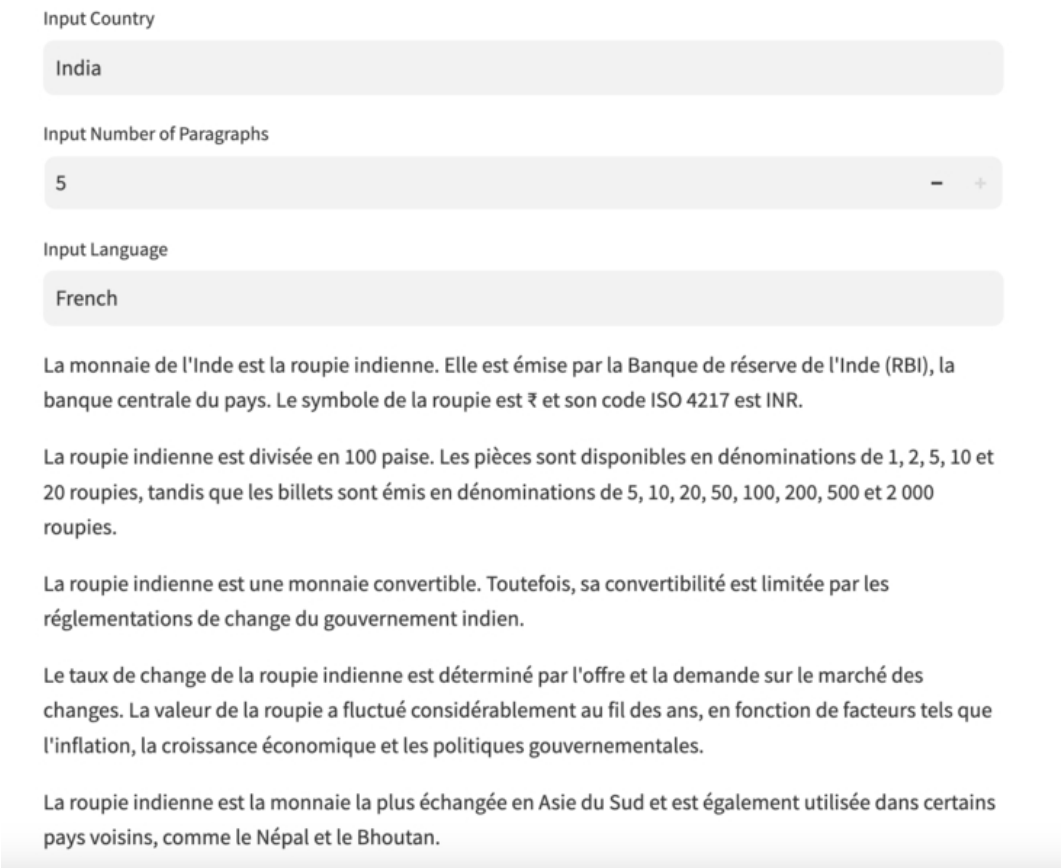

Figure 29. Multiple Inputs Result

Combining the prompt template and Streamlit inputs, you can create a more sophisticated currency information application that can generate an answer exactly N paragraphs long in your preferred language.

Restricting LLM From Answering Unwanted Prompts

Prompt templates can also prevent your model from giving answers to weird questions.

For example, you can ask the LLM about the currency of Narnia, which is a fictional country created by the British author C.S. Lewis:

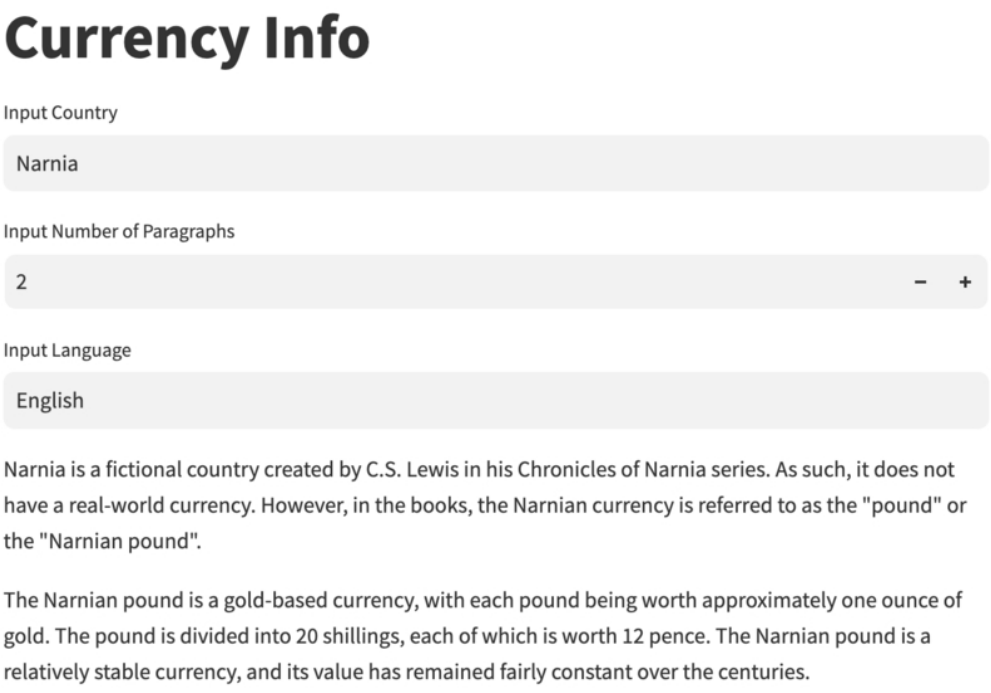

Figure 30. LLM Answering All Kinds of Questions

While the answer is correct, you might not want to give information about fictional or non-existent countries in the application.

Using the prompt template, you can specify the prompt string in multiple lines as shown below:

```
prompt = PromptTemplate(
    input_variables=["country", "paragraph", "language"],
    template='''
    You are a currency expert. You give information about a specific currency
used in a specific country.
    Avoid giving information about fictional places.
    If the country is fictional or non-existent, answer: I don't know.

    Answer the question: What is the currency of {country}?

    Answer in {paragraph} short paragraph in {language}
    ''',
)
```

The prompt above instructs the LLM to not answer when asked about fictional places.

Now if you ask again, the LLM will respond as follows:

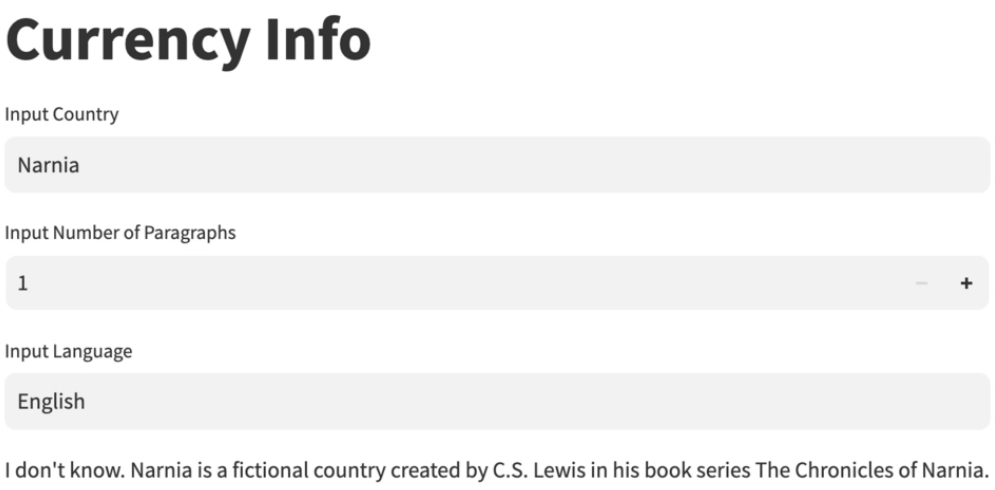

Figure 31. LLM Not Answering

As you can see, the LLM refused to answer when asked about the currency of a fictional country.

With a prompt template, the code is more maintainable and cleaner compared to using f-strings repeatedly.

Summary

The code for this chapter is available in the folder *06_Prompt_Template* from the book source code.

The use of prompt templates enable you to craft a sophisticated instruction for LLMs while requiring only minimal inputs from the user.

The more specific your instruction, the more accurate the response will be.

You can even instruct the LLM to avoid answering unwanted prompts, as shown in the last section.

CHAPTER 7: THE LANGCHAIN EXPRESSION LANGUAGE (LCEL)

In the previous chapter, we called the `prompt.format()` method inside the `llm.invoke()` method as shown below:

```
response = llm.invoke(prompt.format(country=country, paragraph=paragraph,
language=language))
```

While this technique works, LangChain actually provides a declarative way to sequentially execute the `prompt` and `llm` objects.

This declarative way is called the LangChain Expression Language (LCEL for short)

Using LCEL, you can wrap the `prompt` and the `llm` object in a chain as follows:

```
chain = prompt | llm
```

LCEL is marked by the pipe (|) operator, and it can be used to wrap around LangChain components. Components in LangChain include the prompt, the LLM, and the chain itself.

Next, you can call the `invoke()` method from the `chain` object, and pass the inputs required by the `prompt` as a dictionary like this:

```
response = chain.invoke(
  {"country": country, "paragraph": paragraph, "language": language}
)
st.write(response.content)
```

The `chain` object will format the `prompt` and then pass it automatically to the `llm` object.

The `response` object is the same as when you call the `llm.invoke()` method: it's a message object with the answer stored under the `content` attribute.

Sequential Chains

By using LCEL, you can create many chains and send the next prompts once the LLM responds to the previous prompt.

This method of sending the next prompt after the previous prompt has been answered is called the sequential chain.

Based on the input and output results, the sequential chain is divided into 2 categories:

- Simple Sequential Chain
- Regular Sequential Chain

We'll see an example of these sequential chains in the next two sections.

Simple Sequential Chain

The simple sequential chain is where each step in the chain has a single input/ output. The output of one step will be the input of the next prompt:

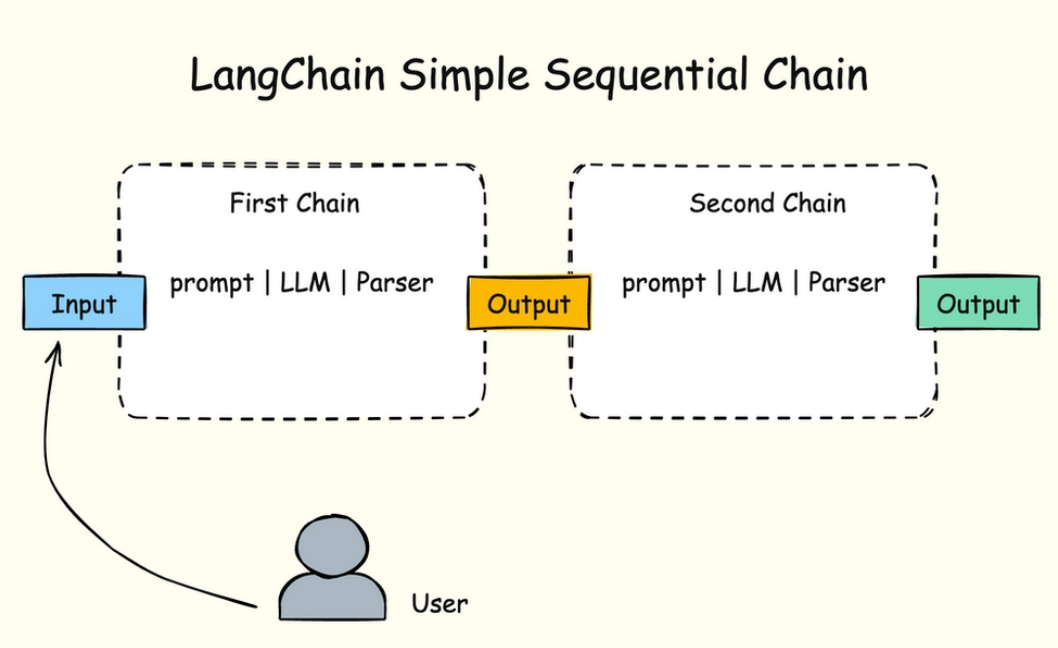

Figure 32. Simple Sequential Chain Illustration

For example, suppose you want to create an application that can write a short essay.

You will provide the topic, and the LLM will first decide on the title, then continue by writing the essay for that topic.

To create the application, you need to create a prompt for the title first:

```
title_prompt = PromptTemplate(
    input_variables=["topic"],
    template="""
    You are an expert journalist.

    You need to come up with an interesting title for the following topic:
{topic}

    Answer exactly with one title
    """,
)
```

The `title_prompt` above receives a single input variable: the `topic` for the title it will generate.

Next, you need to create a prompt for the essay as follows:

```
essay_prompt = PromptTemplate(
    input_variables=["title"],
    template="""
    You are an expert nonfiction writer.

    You need to write a short essay of 350 words for the following title:

    {title}

    Make sure that the essay is engaging and makes the reader feel excited.
    """,
)
```

This `essay_prompt` also takes a single input: the `title` generated by the `title_prompt` which we created before.

Now you need to create two chains, one for each prompt:

```
first_chain = title_prompt | llm | StrOutputParser()
second_chain = essay_prompt | llm
```

The `first_chain` uses the `StrOutputParser()` to parse the LLM response as a string, so you need to import the parser from

LangChain:

```
from langchain_core.output_parsers import StrOutputParser
```

I will show you why the parser is needed below.

For now, you need to combine the chains above into a single overall chain:

```
overall_chain = first_chain | second_chain
```

Now you have a sequential chain made from the `first_chain` and `second_chain` components.

The next step is to update the Streamlit interface and the `if` statement:

```
st.title("Essay Writer")

topic = st.text_input("Input Topic")

if topic:
    response = overall_chain.invoke({"topic": topic})
    st.write(response.content)
```

And you’re finished. If you run the application and ask for a topic, you’ll get a response similar to this:

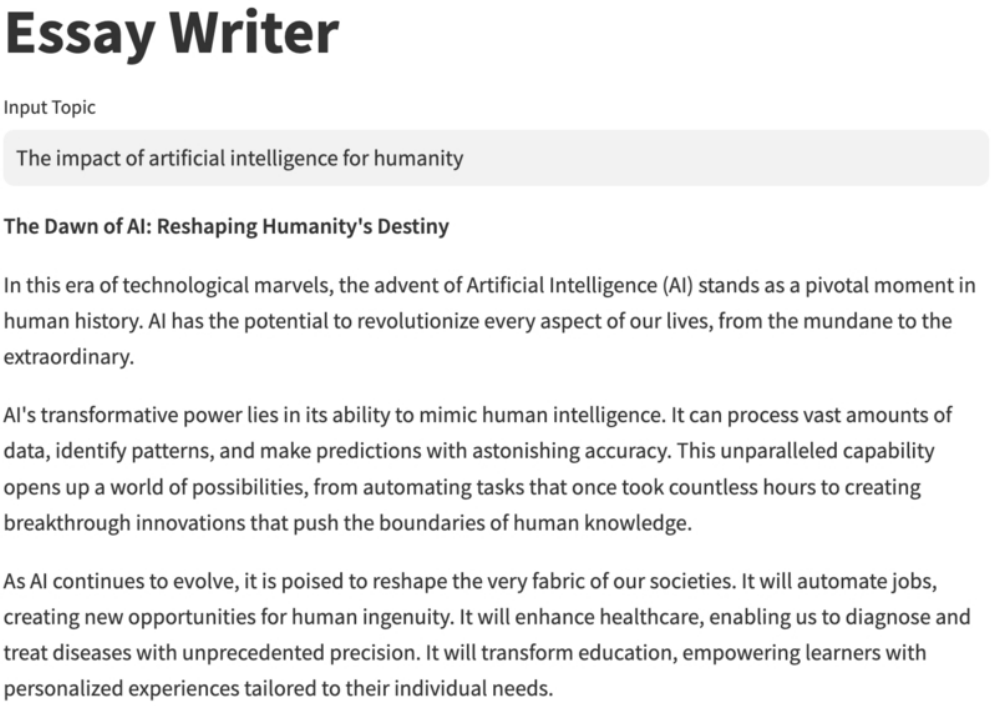

Figure 33. Simple Sequential Chain Result

There are a few paragraphs cut from the result above, but you can already see that the `first_chain` prompt generates the `title` variable used by the `second_chain` prompt.

Using simple sequential chains allows you to break down a complex task into a sequence of smaller tasks, improving the accuracy of the LLM results.

Using Multiple LLMs in Sequential Chain

You can also assign a different LLM for each chain you create using LCEL.

The following example runs the first chain using Google Gemini, while the second chain uses OpenAI GPT:

```
GOOGLE_GEMINI_KEY = config("GOOGLE_GEMINI_KEY")
OPENAI_KEY = config("OPENAI_KEY")

llm = ChatGoogleGenerativeAI(model="gemini-pro",
google_api_key=GOOGLE_GEMINI_KEY)

llm2 = ChatOpenAI(model="gpt-4o", api_key=OPENAI_KEY)

# Use a different LLM for each chain:
first_chain = title_prompt | llm | StrOutputParser()
second_chain = essay_prompt | llm2
```

Because LCEL is declarative, you can swap the components in the chain easily.

Debugging the Sequential Chains

If you want to see the process of sequential chains in more detail, you can enable the debug mode from LangChain `globals` module:

```
from langchain.globals import set_debug

set_debug(True)
```

When you rerun the prompt from Streamlit interface, you'll see the debug output on the command line.

You can see the prompt sent by LangChain by searching for the `[chain/start]` log as follows:

```
[chain/start] [chain:RunnableSequence] Entering Chain run with input:
{
  "topic": "The impact of artificial intelligence for humanity"
}
```

If you search for the second chain input, you'll see the prompt defined as follows:

```
[chain/start] [chain:RunnableSequence > prompt:PromptTemplate] Entering Prompt
run with input:
{
  "input": "**AI's Transformative Embrace: Shaping the Future of Humanity**"
}
```

The input for the second prompt is formatted as a string because we use the `StrOutputParser()` for the first chain.

If you don't parse the output of the first chain, then the second chain prompt will look like this:

```
{
  "prompts": [
    "Human: \n    You are an expert nonfiction writer.\n\n    You need to write
a short essay of 350 words for the following title:\n\n    content=\"**AI's
Transformative Embrace: Shaping the Future of Humanity**\" response_metadata=
{'prompt_feedback': {'block_reason': 0, 'safety_ratings': []}, 'finish_reason':
'STOP', 'safety_ratings': [{'category': 'HARM_CATEGORY_SEXUALLY_EXPLICIT',
'probability': 'NEGLIGIBLE', 'blocked': False}, {'category':
'HARM_CATEGORY_HATE_SPEECH', 'probability': 'NEGLIGIBLE', 'blocked': False},
{'category': 'HARM_CATEGORY_HARASSMENT', 'probability': 'NEGLIGIBLE',
'blocked': False}, {'category': 'HARM_CATEGORY_DANGEROUS_CONTENT',
'probability': 'NEGLIGIBLE', 'blocked': False}]} id='run-6e718aba-b2a6-4dfb-
b9e0-8dc1e338308c-0'\n\n    Make sure that the essay is engaging and make the
reader feel excited."
  ]
}
```

Here, the whole `response` object from the first chain is included in the prompt, such as the `response_metadata`, and `id` attributes.

While the LLM is smart enough to guess the title from the response object, let's minimize this kind of imperfect prompt as much as possible.

Summary

The code for this chapter is available in the folder *07_LCEL* from the book source code.

In this chapter, you've learned about the LangChain Expression Language, which can be used to compose LangChain components in a declarative way.

A chain is simply a wrapper for these LangChain components:

1. The prompt template
2. The LLM to use
3. The parser to process the output from LLM

The components of a chain are interchangeable, meaning you can use the GPT model for the first prompt, and then use the Gemini model for the second prompt, as shown above.

By using LCEL, you can create advanced workflows and interact with Large Language Models to solve a complex task.

In the next chapter, I will show you how to create a regular sequential chain.

CHAPTER 8: REGULAR SEQUENTIAL CHAINS

A regular sequential chain is a more general form of sequential chains that allow multiple inputs and outputs.

The input for the next chain is usually a mix of the output from the previous chain and another source like this:

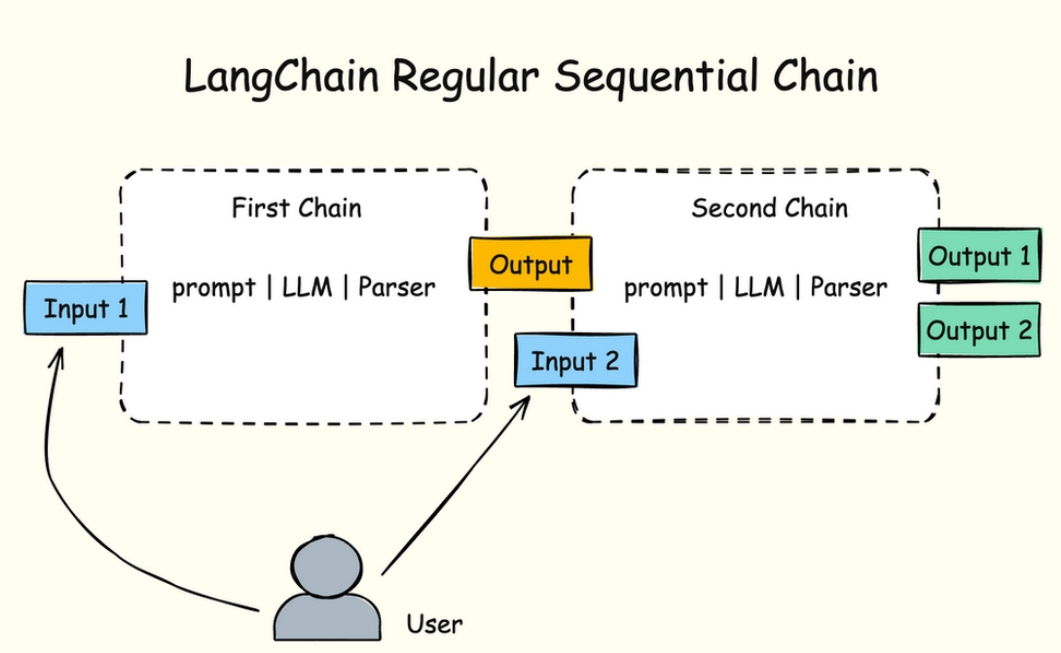

Figure 34. Sequential Chain Illustration

This chain is a little more complicated than a simple sequential chain because we need to track multiple inputs and outputs.

For example, suppose you change the `essay_prompt` from the previous chapter to have two `input_variables` as follows:

```
essay_prompt = PromptTemplate(
    input_variables=["title", "emotion"],
    template="""
    You are an expert nonfiction writer.

    You need to write a short essay of 350 words for the following title:

    {title}

    Make sure that the essay is engaging and make the reader feel {emotion}.
    """,
)
```

The `emotion` input required by the essay prompt doesn't come from the first chain, which only returns the `title` variable.

To inject the `emotion` input, you need to pipe a lambda function between the `first_chain` and `second_chain` when creating the `overall_chain` as follows:

```
overall_chain = (
    first_chain
    | (lambda title: {"title": title, "emotion": emotion})
    | second_chain
)
```

This way, the `title` input is obtained from the output of the `first_chain`, while the `emotion` input is from another source.

You can add a text input for the `emotion` input next:

```
st.title("Essay Writer")

topic = st.text_input("Input Topic")
emotion = st.text_input("Input Emotion")

if topic and emotion:
    response = overall_chain.invoke({"topic": topic})
    st.write(response.content)
```

And now you have multiple inputs, with one input from outside of the first chain result.

Format the Output Variables

A sequential chain usually also tracks multiple output variables.

Advanced LLMs like GPT-4 and Gemini can help us format the output as multiple variables simply by adding a specific instruction in the prompt template.

Back to the `essay_prompt`, you can instruct the LLM to send the output in JSON format:

```
essay_prompt = PromptTemplate(
    input_variables=["title", "emotion"],
    template="""
    You are an expert nonfiction writer.

    You need to write a short essay of 350 words for the following title:

    {title}

    Make sure that the essay is engaging and makes the reader feel {emotion}.

    Format the output as a JSON object with three keys: 'title', 'emotion',
'essay' and fill them with respective values
    """,
)
```

Now that the LLM is instructed to format the output as a JSON object, you need to parse the output using the `JsonOutputParser`.

Pipe the `JsonOutputParser` to the `second_chain` as shown below:

```
from langchain_core.output_parsers import StrOutputParser, JsonOutputParser

# ...

first_chain = title_prompt | llm | StrOutputParser()
second_chain = essay_prompt | llm | JsonOutputParser()
```

Next, change the `st.write()` method to write the `response` object directly:

```
if topic and emotion:
    response = overall_chain.invoke({"topic": topic})
    st.write(response)
```

Now you can run the application and see that the output is indeed in JSON format:

Figure 35. Receiving JSON Output

If you want to write each output variable, you can access them directly as follows:

```
if topic and emotion:
    response = overall_chain.invoke({"topic": topic})
    st.write(response.title)
    st.write(response.emotion)
    st.write(response.essay)
```

Now you have a sequential chain that tracks multiple inputs and outputs. Nice work!

Summary

The code for this chapter is available in the folder *08_Sequential_Chain* from the book source code.

When you create a sequential chain, you can add extra inputs to the next chain that are not sourced from the previous chain.

Using a lambda function, you can add custom logic before running the chain, enabling you to send previous chain results together with extra inputs.

You can also format the output as a JSON object to make the response better organized and easier to process.

CHAPTER 9: IMPLEMENTING CHAT HISTORY IN LANGCHAIN

So far, the LLM took the question we asked it and gave an answer retrieved from the training data.

Going back to the simple question and answer application in Chapter 5, you can try to ask the LLM a question such as:

1. When was the last FIFA World Cup held?
2. Multiply the year by 2

At the time of this writing, the last FIFA World Cup was held in 2022. Reading the second prompt above, we can understand that the 'year' refers to '2022'.

However, because the LLM has no awareness of the previous interaction, the answer won’t be related.

With GPT, the LLM refers to the current year instead of the last FIFA World Cup year:

Figure 36. LLM Out of Context Example

The LLM can't understand that we are making a follow-up instruction to the previous question.

To address this issue, you need to save the previous messages and use them when sending a new prompt.

To follow along with this chapter, you can copy the code from Chapter 5 and use it as a starter.

Creating a Chat Prompt Template

First, you need to create a chat prompt template that has the chat history injected into it.

A chat prompt template is different from the usual prompt template. It accepts a list of messages, and each message can be associated with a specific role.

Here's an example of a chat prompt template:

```
from langchain_core.prompts import ChatPromptTemplate

chat_template = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful AI bot. Your name is {name}."),
```

```
        ("human", "Hello, how are you doing?"),
        ("ai", "I'm doing well, thanks!"),
        ("human", "{user_input}"),
    ]
)
```

In the example above, the messages are associated with the 'system', 'human', and 'ai' roles.

You can use the `ChatPromptTemplate` and `MessagesPlaceholder` classes to create a prompt that accepts a chat history as follows:

```
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are an AI chatbot having a conversation with a human. Use the
following context to understand the human question. Do not include emojis in
your answer. ",
        ),
        MessagesPlaceholder(variable_name="chat_history"),
        ("human", "{input}"),
    ]
)
```

The `MessagesPlaceholder` class acts as an open parameter from which you can inject the chat history, which is what we're going to do next.

Saving Messages in Streamlit Chat History

The Streamlit session state is used to store variables as long as the Streamlit application is running.

To save chat history in Streamlit, you need to import the `StreamlitChatMessageHistory` class, and instantiate an object from that class:

```
from langchain_community.chat_message_histories import
StreamlitChatMessageHistory

# other code ...

chain = prompt | llm

history = StreamlitChatMessageHistory()
```

Next, you need to inject the history stored in Streamlit into the chain.

This can be done by using the `RunnableWithMessageHistory` class, which is a wrapper around a chain that injects the chat history for you:

```
from langchain_core.runnables.history import RunnableWithMessageHistory

# other code...

chain = prompt | llm

history = StreamlitChatMessageHistory()

chain_with_history = RunnableWithMessageHistory(
    chain,
    lambda session_id: history,
    input_messages_key="input",
    history_messages_key="chat_history",
)
```

When creating the `RunnableWithMessageHistory` object, you need to pass the chain that you want to inject history into and a function that returns the chat history.

The `input_messages_key` is the input key that exists in the prompt, while `history_messages_key` is the variable that accepts the history (it should be the same as the `variable_name` passed to `MessagesPlaceholder` before)

Now that the `chain_with_history` object is completed, you can modify the Streamlit interface and the `if` statement as follows:

```
st.title("Q & A With AI")

question = st.text_input("Your Question")
if question:
    response = chain_with_history.invoke(
        {"input": question}, config={"configurable": {"session_id": "any"}}
    )
    st.write(response.content)
```

When invoking the `chain_with_history` object, you need to pass the `session_id` key into the `config` parameter as shown above.

The `session_id` can be of any string value.

Now you can test the application by giving a follow-up question:

```
Q: When was the last FIFA World Cup held?
A: The last FIFA World Cup was held in 2022.
Q: Multiply the year by 2
A: 2022 multiplied by 2 is 4044.
```

Because the chat history is injected into the prompt, the LLM can put the second question in the context of the first.

If you activate the debug mode with `set_debug(True)`, you'll see the prompt as shown below:

```
"prompts": [
    "System: You are an AI chatbot having a conversation with a human. Use the
following context to understand the human question. Do not include emojis in
your answer. \nHuman: When was the last FIFA World Cup held?\nAI: The last FIFA
World Cup was held in 2022. \n\nHuman: Multiply the year by 2"
  ]
```

As you can see, the `RunnableWithMessageHistory` injects the history right where you put the `MessagesPlaceholder()` in the prompt.

Showing Chat History

To show the chat history, you need to add a `for` loop below the `st.write()` method:

```
st.write(response.content)
for message in st.session_state["langchain_messages"]:
    if message.type == "human":
        st.write("Question: " + message.content)
    else:
        st.write("Answer: " + message.content)
```

The `for` loop above iterates over the `st.session_state["langchain_messages"]` list.

We also check on the `type` of the message to prepend the 'Question' and 'Answer' string before the message `content`.

Now you should see the chat history as follows:

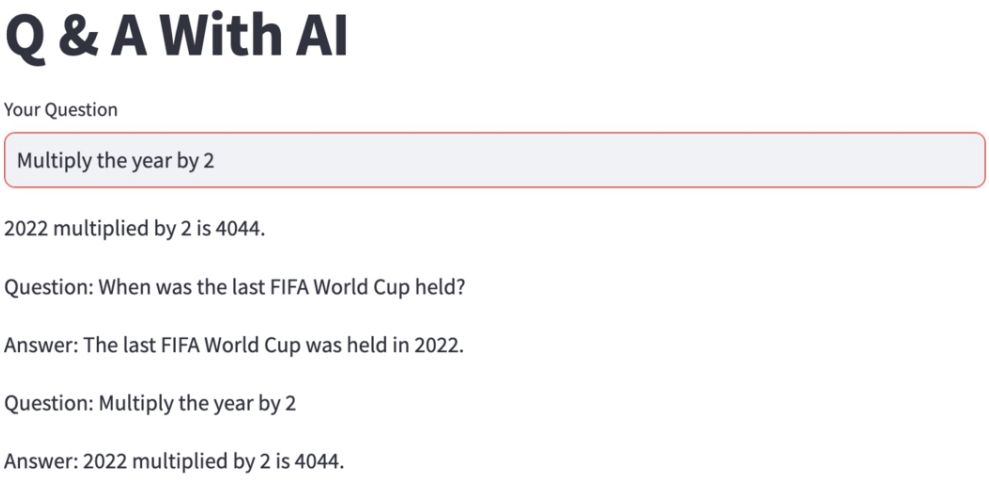

Figure 37. Chat History Shown

Nice work so far!

Cloning ChatGPT With Streamlit Chat Input

Now that we've added chat history to our application, we can create an interface that looks like ChatGPT with the help of Streamlit.

Instead of using `text_input` for the text box, you need to use `chat_input` as follows:

```
question = st.chat_input("Your Question")
```

Next, use the with `st.chat_message()` method and pass the role 'user' to write down the question:

```
question = st.chat_input("Your Question")
if question:
    with st.chat_message("user"):
        st.markdown(question)
```

Then replace the `invoke()` call with `stream()` as follows:

```
if question:
    with st.chat_message("user"):
        st.markdown(question)
    response = chain_with_history.stream(
        {"input": question}, config={"configurable": {"session_id": "any"}}
    )
```

When writing the response, you need to use with `st.chat_message()` and pass the 'assistant' role to the method:

```
   response = chain_with_history.stream(
        {"input": question}, config={"configurable": {"session_id": "any"}}
    )
    with st.chat_message("assistant"):
        st.write_stream(response)
```

Finally, you need to show the chat history just below the page title as follows:

```
st.title("Q & A With AI")

for message in st.session_state["langchain_messages"]:
    role = "user" if message.type == "human" else "assistant"
    with st.chat_message(role):
        st.markdown(message.content)
```

You can determine the `role` of the message by checking the `message.type` value.

Run the application again, and you will see the output as shown below:

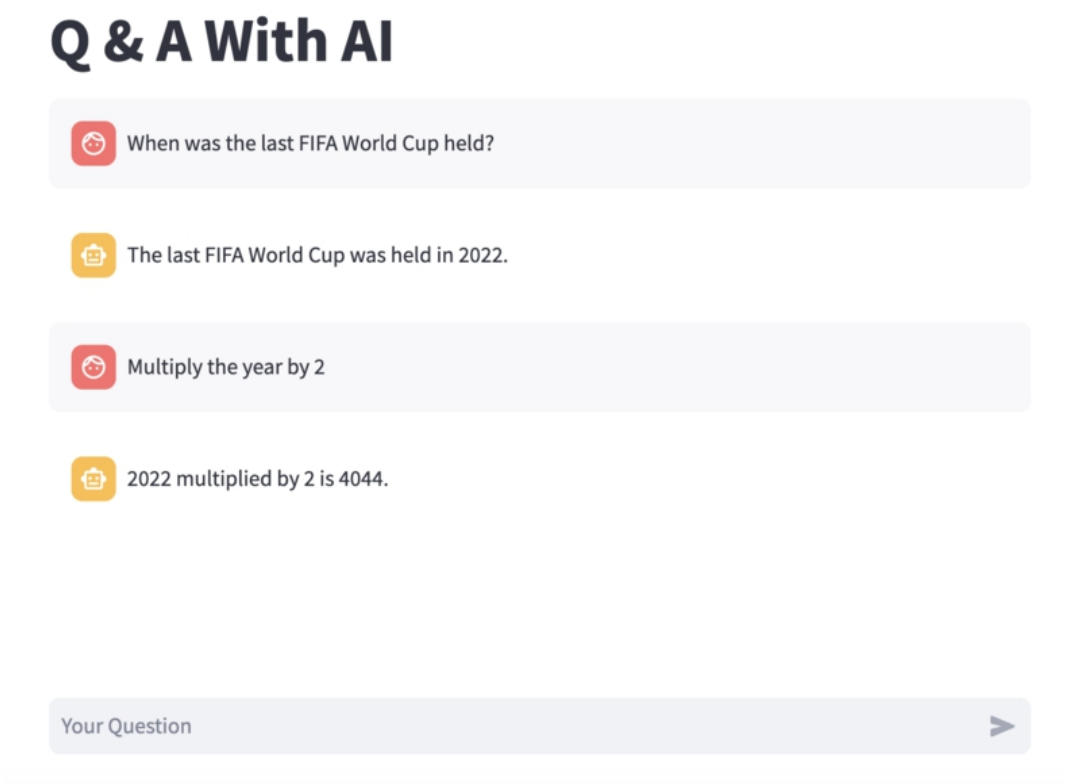

Figure 38. A Simpler ChatGPT Clone

Now you have a proper AI chat application that looks just like ChatGPT. Awesome!

Summary

The code for this chapter is available in the folder *09_Chat_History* from the book source code.

In this chapter, you've learned how to inject chat history into the prompt using LangChain's `RunnableWithMessageHistory` class.

Adding the chat history enables AI to contextualize your question based on the previous messages.

Combining chat history with Streamlit user interface, you can create an application similar to ChatGPT easily.

CHAPTER 10: AI AGENTS AND TOOLS

An AI agent is a thinking software that is capable of solving a task through a sequence of actions. It uses the LLM as a reasoning engine to plan and execute actions.

All you need to do is to give the agent a specific task. The agent will process the task, determine the actions needed to solve it, and then take those actions.

An agent can also use tools to take actions in the real-world, such as searching for specific information on the internet.

Here’s an illustration to help you understand the concept of agents:

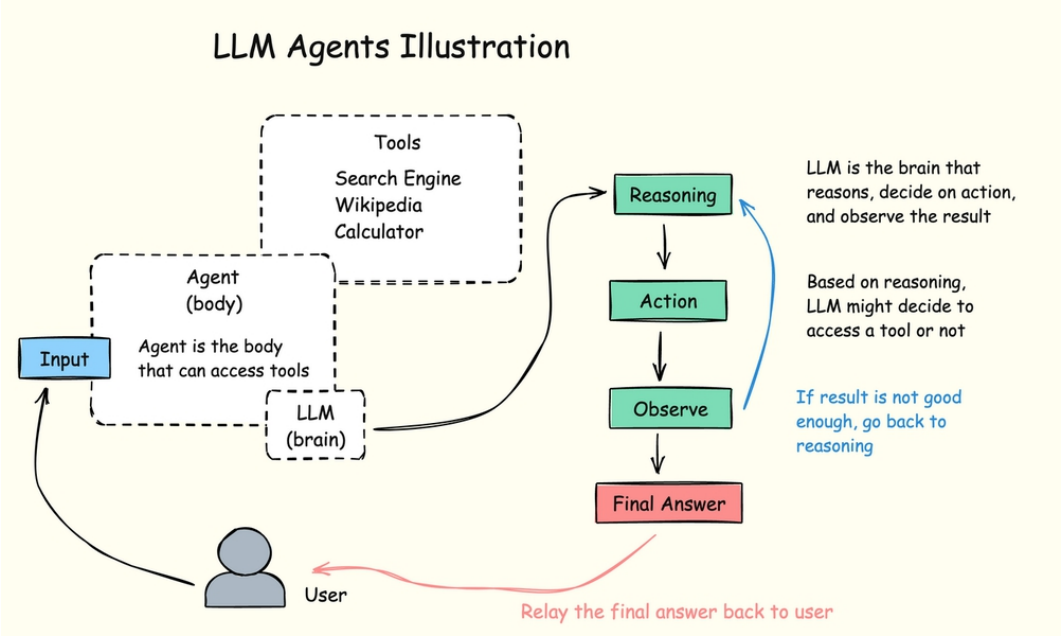

Figure 39. LLM Agents Illustration

Not all LLMs are capable of creating an agent, so advanced models like GPT-4, Gemini 1.5 Pro, or Mistral are required.

Let me show you how to create an agent using LangChain next.

Creating an AI Agent

Create a new Python file named `react_agent.py` and import the following modules:

```
from decouple import config
from langchain import hub
from langchain.agents import create_react_agent, AgentExecutor
from langchain_community.agent_toolkits.load_tools import load_tools
from langchain_google_genai import ChatGoogleGenerativeAI
```

The `config` and `ChatGoogleGenerativeAI` have already been used before, but the rest are new modules used to create an AI agent.

The `hub` module is used to retrieve a prompt from the LangChain community hub. You can visit the hub at https://smith.langchain.com/hub

The LangChain community hub is an open collection of prompts that you can use for free in your projects.

The `create_react_agent` module creates an agent that uses ReAct prompting, while `AgentExecutor` manages the execution of the agent, such as processing inputs, generating responses, and updating the agent's state.

The `load_tools` module is used to load tools available in LangChain.

Next, initialize the `llm` and get the prompt from the hub as follows:

```
OPENAI_KEY = config("OPENAI_KEY")

llm = ChatOpenAI(model="gpt-4o", api_key=OPENAI_KEY)

prompt = hub.pull("hwchase17/react")
```

The `hub.pull()` method retrieves the prompt from the repository in the LangChain hub. Here, we use the "react" prompt created by the user "hwchase17".

If you want to see the prompt, you can visit https://smith.langchain.com/hub/hwchase17/react

Next, load the tools we want to provide to the LLM, then create the agent executor object as follows:

```
tools = load_tools(["wikipedia", "ddg-search", "llm-math"], llm)

agent = create_react_agent(llm, tools, prompt)

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

There are three tools we provide to the agent:

1. `wikipedia` for accessing and summarizing Wikipedia articles
2. `ddg-search` for searching the internet using the DuckDuckGo search engine
3. `llm-math` for calculating mathematical equations

To run these tools properly, you need to install the packages using pip:

```
pip install wikipedia==1.4.0 duckduckgo-search==6.1.2 numexpr==2.10.0
```

The `numexpr` package is used by `llm-math` behind the scene.

Once the installation is finished, finish the agent by adding an `input` prompt and call the `invoke()` method:

```
question = input("Give me a task: ")
agent_executor.invoke({"input": question})
```

And the AI agent is now completed.

You can run the agent using Python as follows:

```
python react_agent.py
```

Now give a task for the agent to finish, such as 'Who was the first president of America?'

Because the `verbose` parameter is set to `True` in `AgentExecutor`, you will see the reasoning done by the LLM:

```
Give me a task: Who was the first president of America?


> Entering new AgentExecutor chain...
Thought: I should use wikipedia to find the first president of America.
Action: wikipedia
Action Input: first president of the United States of AmericaPage: List of presidents of the United States
```

Figure 40. LLM Agent Reasoning and Taking Actions

The agent will perform several actions before it reaches the final answer:

```
Article II of the Constitution establishes the executive branch of the federal government and vests execut
ive power in the president. The power includes the execution and enforcemeThought: I can see from the wiki
pedia article that George Washington was the first president.
Final Answer: George Washington


> Finished chain.
```

Figure 41. LLM Agent Finished

Now these outputs are shown because we use the verbose parameter.

You can show the steps the agent takes by calling the `stream()` method instead of `invoke()`, then check the returned response for a specific key:

```
# Turn verbose off to avoid polluting the command line
agent_executor = AgentExecutor(agent=agent, tools=tools)

question = input("Give me a task: ")
# Call stream() instead of invoke()
response = agent_executor.stream({"input": question})
# Agent Action
if "actions" in response:
    for action in response["actions"]:
        print(f"Calling Tool: `{action.tool}` with input
```

```
`{action.tool_input}`")
# Observation
elif "steps" in response:
    for step in response["steps"]:
        print(f"Tool Result: `{step.observation}`")
# Final result
elif "output" in response:
    print(f'Final Output: {response["output"]}')
else:
    raise ValueError()
print("---")
```

Now if you run the agent again, you will only be shown the actions, steps, and final output. The summaries are omitted for brevity below:

```
Give me a task: Who was the first president of America?
Calling Tool: `wikipedia` with input `first president of the United States`
---
Tool Result: `Page: George Washington
Summary: ...

Page: President of the United States
Summary: ...
---
Calling Tool: `wikipedia` with input `first president of the United States`
---
Tool Result: `Page: George Washington
Summary: ...

Page: President of the United States
Summary: ...
---
Final Output: George Washington was the first president of the United States.
---
```

If you use another model like GPT-4o, you might get the final output directly:

```
Give me a task: Who was the first president of America?
Final Output: The first president of America was George Washington.
---
```

This is because the LLM has the answer in its training data, so it decides to answer directly.

Adding Streamlit UI for the Agent

Now that the agent is running, you can add the Streamlit UI to improve the user experience.

You only need to add a single `chat_input` to accept a task, then write the question and answer as follows:

```
import streamlit as st

# ... other code

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

st.title("AI Agent")

question = st.chat_input("Give me a task: ")
if question:
    with st.chat_message("user"):
        st.markdown(question)
    with st.chat_message("assistant"):
        for response in agent_executor.stream({"input": question}):
            # Agent Action
            if "actions" in response:
                for action in response["actions"]:
                    st.write(f"Calling Tool: `{action.tool}` with input
`{action.tool_input}`")
            # Observation
            elif "steps" in response:
                for step in response["steps"]:
                    st.write(f"Tool Result: `{step.observation}`")
            # Final result
            elif "output" in response:
                st.write(f'Final Output: {response["output"]}')
            else:
                raise ValueError()
            st.write("---")
```

Because the output is written on the Streamlit UI, it's fine to turn on the `verbose` parameter in the `AgentExecutor` for development.

Run the application using Streamlit:

```
streamlit run app.py
```

You can now ask different kinds of questions to see if the LLM can use the available tools.

If you ask 'What is the weather in London today?', the LLM should use DuckDuckGo search to seek the latest weather information in London:

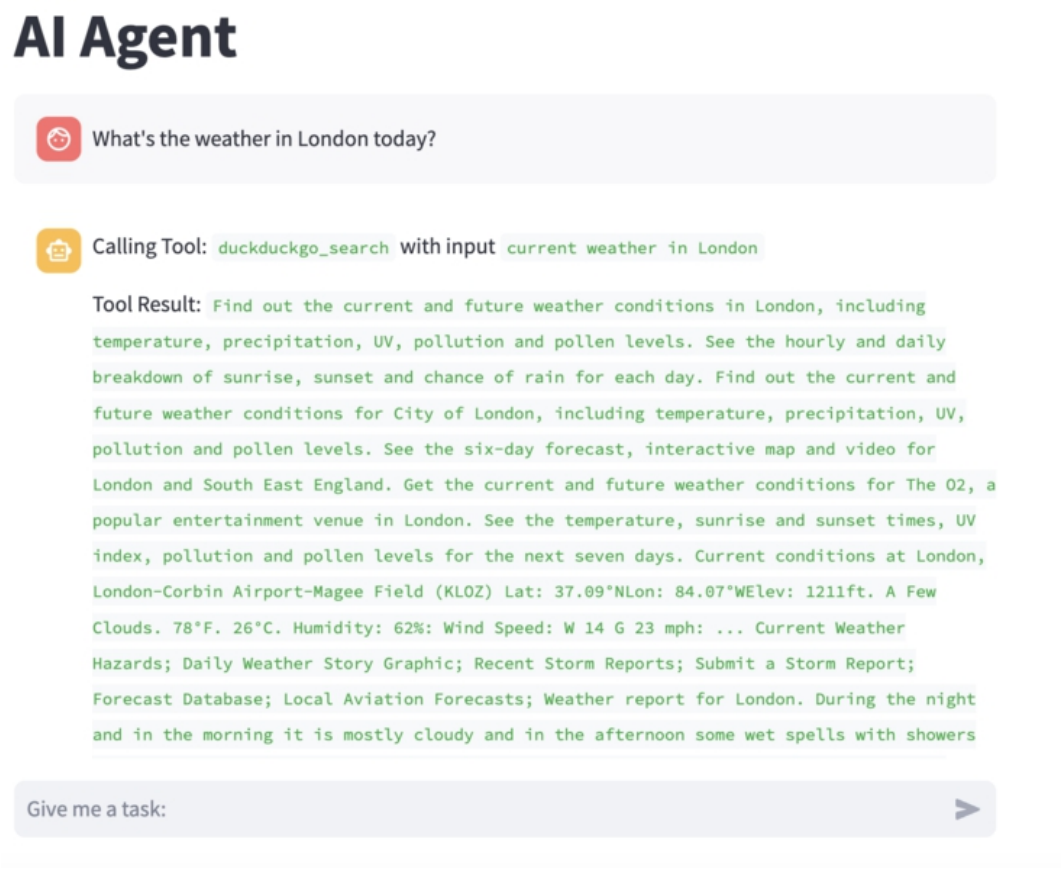

Figure 42. LLM Agent Doing Search

If you ask a math question such as 'Take 5 to the power of 2 then multiply that by the sum of six and three', the agent should use the `llm-math` tool:

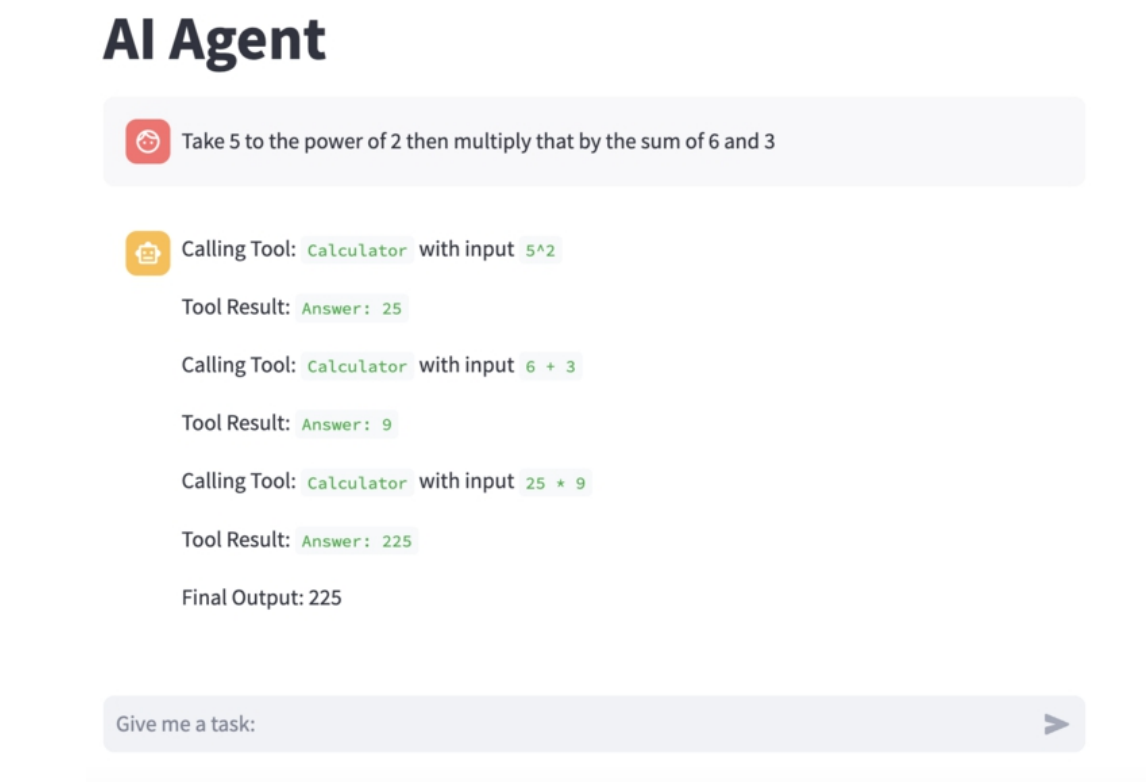

Figure 43. LLM Agent Doing Math

What's more, the AI agent can also use many tools to complete a task.

If you ask 'When was Harry Potter first published? Add 2 to the year', then the agent will perform the following steps:

```
Give me a task: when was Harry Potter first published? Add 2 to the year
Calling Tool: `wikipedia` with input `Harry Potter publication date`
---
Tool Result: `Page: Harry Potter and the Order of the Phoenix
Summary: ...

Page: Harry Potter and the Philosopher's Stone
Summary: ...
```

```
Page: Harry Potter and the Half-Blood Prince
Summary: ...
---
Calling Tool: `Calculator` with input `1997 + 2`
---
Tool Result: `Answer: 1999`
---
Final Output: 1999.
```

Here, the agent uses the Wikipedia and LLM-math tools to finish the job. Very nice!

List of Available AI Tools

An AI agent can only use the tools you added when you create the agent.

The list of tools available in LangChain can be found on the `load_tools` module.

On the `import load_tools` line, press F12 while placing your cursor somewhere inside `load_tools`.

You can also use the mouse by right-clicking the `load_tools` name and select 'Go To Definition' from the context menu

On the `load_tools.py` file, search for the `get_all_tool_names()` function as shown below:

```
def get_all_tool_names() -> List[str]:
    """Get a list of all possible tool names."""
    return (
        list(_BASE_TOOLS)
        + list(_EXTRA_OPTIONAL_TOOLS)
        + list(_EXTRA_LLM_TOOLS)
        + list(_LLM_TOOLS)
```

```
        + list(DANGEROUS_TOOLS)
    )
```

The *_TOOLS constants above contains the name of the tool you can use.

For example, the DuckDuckGo search is found at _EXTRA_OPTIONAL_TOOLS as shown below:

Figure 44. LLM Agent Available Tools

You can also see other tools like Google search and Bing search, but these tools require an API key to run.

Types of AI Agents

There are several types of AI agents identified today, and the one we created is called a ReAct (Reason + Act) agent.

The ReAct agent is a general-purpose agent, and there are more specialized agents such as the XML agent and JSON agent.

You can read more about the different agent types at https://python.langchain.com/v0.1/docs/modules/agents/agent_types/

As LLM and LangChain improve, new types of agents might be created, so the definitions above won't always be relevant.

Summary

The code for this chapter is available in the folder *10_Agents_and_Tools* from the book source code.

While we don't have an autonomous robot helper (yet) in our world today, we can already see how the development of AI agents might one day be used as the brains of AI robots.

AI agents are an amazing innovation that shows how a machine can come up with a sequence of actions to take to accomplish a goal.

When you create an agent, the LLM is used as a reasoning engine that needs to come up with steps of logic to accomplish a task.

The agent can also use various tools to act, such as browsing the web or solving a math equation.

More complex tasks that use multiple tools can also be executed by these agents.

Sometimes, a well-trained LLM can answer directly from the training data, bypassing the need to use tools.

CHAPTER 11: INTERACTING WITH DOCUMENTS IN LANGCHAIN

One of the most interesting AI use cases is the Chat With Document feature, which enables users to interact with a document using conversational queries.

By simply asking questions, users can quickly find relevant information, summarize content, and gain insights without having to read and sort through pages of text.

Here’s an illustration of the process required to create a Chat With Document application:

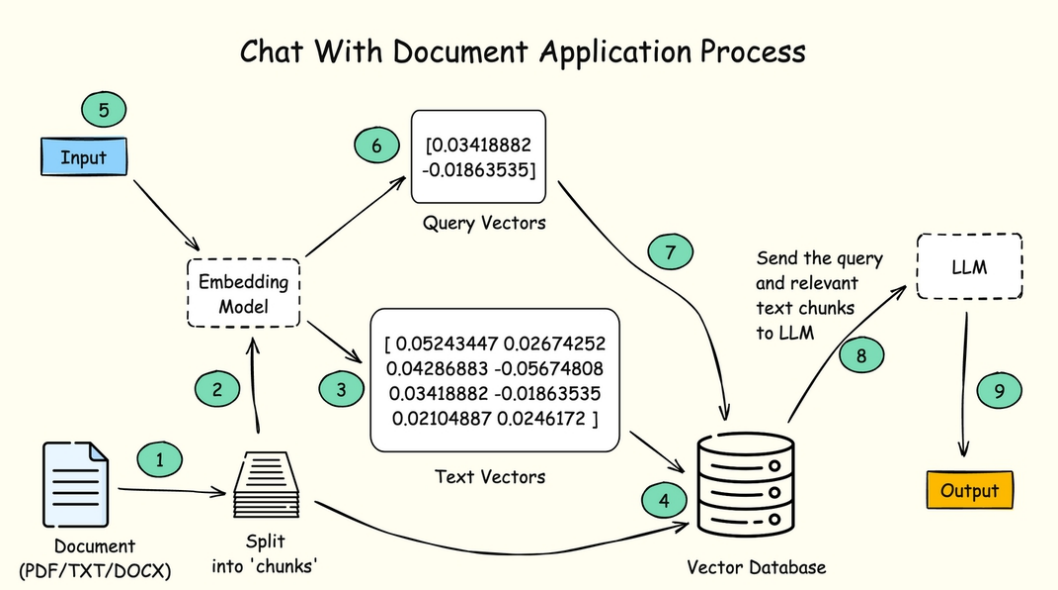

Figure 45. Chat With Document Application Process

You first need to split a single document into chunks so that the document can be processed and indexed effectively. The chunks are then converted into vector embeddings.

Vector embeddings are arrays of numbers that represent information of various types, including text, images, audio, and more, by capturing their features in a numerical format.

When you give a question, LangChain will convert the query into vectors, and then search the document vectors for the most relevant matches.

LangChain then sends the user query and relevant text chunks to LLM so that the LLM can generate a response based on the given input.

The process of finding relevant text information and sending it to the LLM is also known as Retrieval Augmented Generation, or RAG for short.

Using LangChain and Streamlit, you can create an application that allows you to upload a document and ask questions relevant to the content of that document.

Let's jump in.

Getting the Document

You're going to upload a document to the application and ask questions about it. You can get the text file named `ai-discussion.txt` from the *11_Chat_With_Document* folder.

The text file contains a fictional story that discusses the impact of AI on humanity.

Because it's fictional, you can be sure that the LLM obtains the answer from the document and not from its training data.

Installing ChromaDB

To create a Chat With Document application, you need to install a package called ChromaDB:

```
pip install chromadb == 0.4.18
```

ChromaDB is the vector database we're going to use to store and query vector data.

Building the Chat With Document Application

Create a new Python file named `rag_prompt.py`, then import the packages required for the Chat With Document application as follows:

```
from decouple import config
from langchain_openai import ChatOpenAI, OpenAIEmbeddings
from langchain_core.prompts import ChatPromptTemplate
# New packages:
from langchain_community.document_loaders import TextLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain.chains import create_retrieval_chain
```

The `OpenAIEmbeddings` class is used to access the OpenAI embedding model.

The `TextLoader` class is used to load a text file, while `RecursiveCharacterTextSplitter` is used to split a text into small chunks so that it can be processed more efficiently by LLMs.

The `Chroma` class is used to create the vector database. The `create_retrieval_chain` retrieves a document and passes it to the `create_stuff_documents_chain`, which passes the document to the LLM.

With the packages imported, you can define the `llm` next as shown below:

```
OPENAI_KEY = config("OPENAI_KEY")

llm = ChatOpenAI(model="gpt-4o", api_key=OPENAI_KEY)
```

After the `llm`, load the text using `TextLoader` and pass the path to the document in `TextLoader` as follows:

```
loader = TextLoader("./ai-discussion.txt")
documents = loader.load()
```

With the list of `documents` loaded, you need to create the text splitter and split the document into chunks:

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=1000,
chunk_overlap=200)
chunks = text_splitter.split_documents(documents)
```

To convert the document `chunks` into vectors, you need to use an embedding model.

OpenAI provides an API endpoint to turn documents into vectors, and LangChain provides the `OpenAIEmbeddings` class so that you can use this embedding easily.

Add the following code below the `chunks` variable:

```
embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_KEY)

vector_store = Chroma.from_documents(chunks, embeddings)

retriever = vector_store.as_retriever()
```

The `Chroma.from_documents()` method sends the `chunks` and `embeddings` data to the Chroma database.

Then, you need to call the `as_retriever()` method to create a `retriever` object, which accepts the user input and returns relevant chunks of the document.

Next, you need to create the prompt to send to the LLM:

```
system_prompt = (
    "You are an assistant for question-answering tasks. "
    "Use the following pieces of retrieved context to answer "
    "the question. If you don't know the answer, say that you "
    "don't know. Use three sentences maximum and keep the "
    "answer concise."
    "\n\n"
```

```
    "{context}"
)

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", system_prompt),
        ("human", "{input}"),
    ]
)
```

With the prompt created, you need to create a chain by calling the `create_stuff_documents_chain()` function and pass `llm` and `prompt` like this:

```
question_answer_chain = create_stuff_documents_chain(llm, prompt)
```

This `question_answer_chain` will handle filling the prompt template and sending the prompt to the LLM.

Next, pass the `question_answer_chain` to the `create_retrieval_chain()` function:

```
rag_chain = create_retrieval_chain(retriever, question_answer_chain)
```

The `rag_chain` above will pass the input to the `retriever`, which returns a relevant part of the document to the chain.

The relevant part of the chain and the input are then passed to the `question_answer_chain` to get the result.

Now you can ask for user input, then run the `invoke()` method with that input:

```
question = input("Ask Your Question: ")
if question:
    response = rag_chain.invoke({"input": question})
    print(response["answer"])
```

When the LLM returns the answer, you print the answer to the command line.

Now the application is completed. You can run it using Python from the command line:

```
python rag_prompt.py
```

And then ask a question relevant to the document context. Here's an example:

```
python3 rag_prompt.py
Ask Your Question: Where does Mr. Thompson work?
Mr. Thompson works as the Chief AI Scientist at VegaTech Inc.
```

As we can see, the LLM can answer the question by analyzing the prompt created by LangChain and our input.

Adding Streamlit Chat UI to the Application

With the core functionality of the application finished, we can add a web UI and chat memory to improve the application.

Let's add the Web UI first. Import Streamlit into the application, then add the Streamlit UI below the `rag_chain` variable:

```
import streamlit as st

# ... other code
rag_chain = create_retrieval_chain(retriever, question_answer_chain)

st.title("Chat With Document")

question = st.chat_input("Your Question: ")
if question:
    with st.chat_message("user"):
        st.markdown(question)
    response = rag_chain.invoke({"input": question})
```

```
    with st.chat_message("assistant"):
        st.write(response["answer"])
```

Now you can run the application using the `streamlit run` command.

The next step is to add the memory part to the application.

Adding Chat Memory for Context

When adding a chat memory to a RAG chain, you also need to upgrade the `retriever` object by creating a history-aware retriever.

This history-aware retriever is then used to contextualize your latest question by analyzing the chat history.

You need to import the `create_history_aware_retriever` chain, then create a `history_aware_retriever` object as follows:

```
from langchain.chains import create_history_aware_retriever

# ... other code

retriever = vector_store.as_retriever()

contextualize_system_prompt = """Given a chat history and the latest user
question \
which might reference context in the chat history, formulate a standalone
question \
which can be understood without the chat history. Do NOT answer the question, \
just reformulate it if needed and otherwise return it as is."""
contextualize_prompt = ChatPromptTemplate.from_messages(
    [
        ("system", contextualize_system_prompt),
        MessagesPlaceholder("chat_history"),
        ("human", "{input}"),
    ]
)
```

```
history_aware_retriever = create_history_aware_retriever(
    llm, retriever, contextualize_prompt
)
```

After creating the `history_aware_retriever` object, you need to update the `rag_chain` to use the retriever as follows:

```
rag_chain = create_retrieval_chain(history_aware_retriever,
question_answer_chain)
```

Now that you have an updated RAG chain, pass the chain to the `RunnableWithMessageHistory()` class like we did in Chapter 9:

```
# Add the imports
from langchain_community.chat_message_histories import
StreamlitChatMessageHistory
from langchain_core.runnables.history import RunnableWithMessageHistory

# ... other code

rag_chain = create_retrieval_chain(history_aware_retriever,
question_answer_chain)

history = StreamlitChatMessageHistory()

conversational_rag_chain = RunnableWithMessageHistory(
    rag_chain,
    lambda session_id: history,
    input_messages_key="input",
    history_messages_key="chat_history",
    output_messages_key="answer",
)
```

To show the chat history to users, look for the `message` object in `st.session_state` like before:

```
st.title("Chat With Document")

for message in st.session_state["langchain_messages"]:
    role = "user" if message.type == "human" else "assistant"
    with st.chat_message(role):
        st.markdown(message.content)
```

As the last step, change the chain invoked when the application receives a question, and add the `config` parameter for the "session_id":

```
if question:
    with st.chat_message("user"):
        st.markdown(question)
    response = conversational_rag_chain.stream(
        {"input": question}, config={"configurable": {"session_id": "any"}}
    )
    with st.chat_message("assistant"):
        st.write(response["answer"])
```

Okay. Now we have a nice interface for interacting with the document, and the AI is aware of the chat history.

Streaming the Answer

So far, we still use the `invoke()` method to run the chain. Let's try to stream the AI answer instead:

```
with st.chat_message("user"):
    st.markdown(question)
response = conversational_rag_chain.stream(
    {"input": question}, config={"configurable": {"session_id": "any"}}
)
with st.chat_message("assistant"):
    st.write_stream(response)
```

If you run the application and pass a question to the LLM, you'll notice that the `response` is a dictionary with four keys: "input", "chat_history", "context", and "answer".

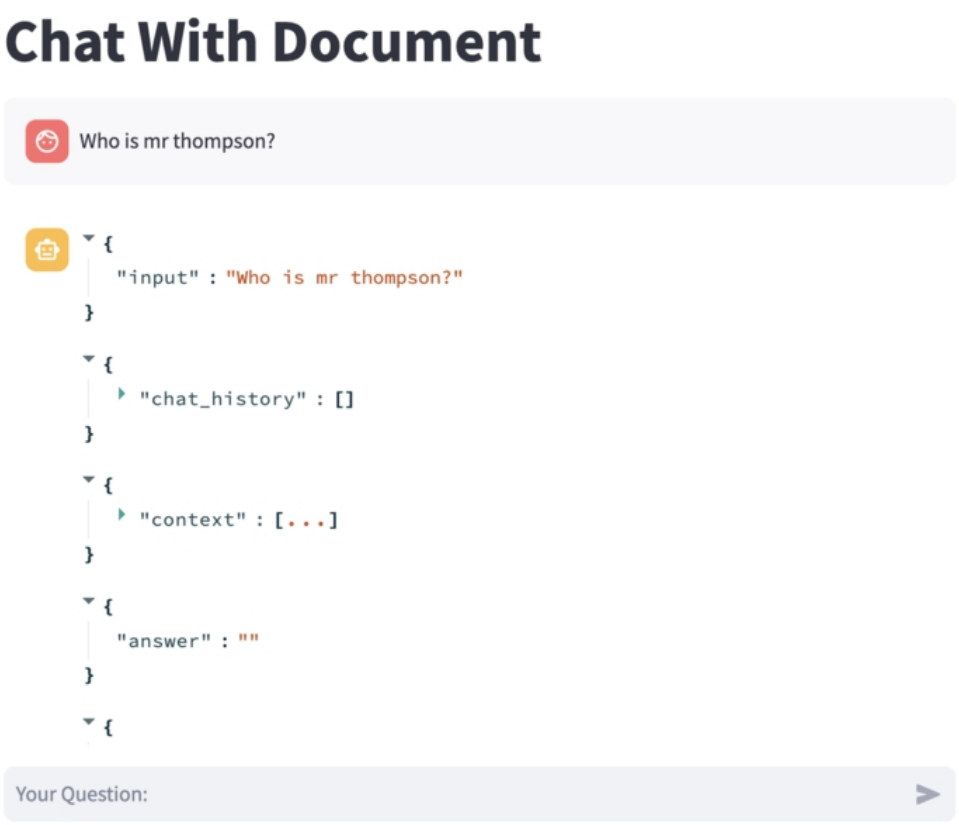

Figure 46. Stream Response From RAG Chain

If you decide to `write_stream()` only the "answer" key, you'll get a 'generator' object is not subscriptable TypeError from Python:

```
st.write_stream(response["answer"]) # TypeError
```

To stream only the answer to the user, you need to call the `pick()` method and pass the "answer" key as follows:

```
answer_chain = conversational_rag_chain.pick("answer")
response = answer_chain.stream(
    {"input": question}, config={"configurable": {"session_id": "any"}}
)
```

The `pick()` method filters the returned output and only returns the content of the key you passed as its argument.

This way, the `response` object is now a stream that returns only the answer:

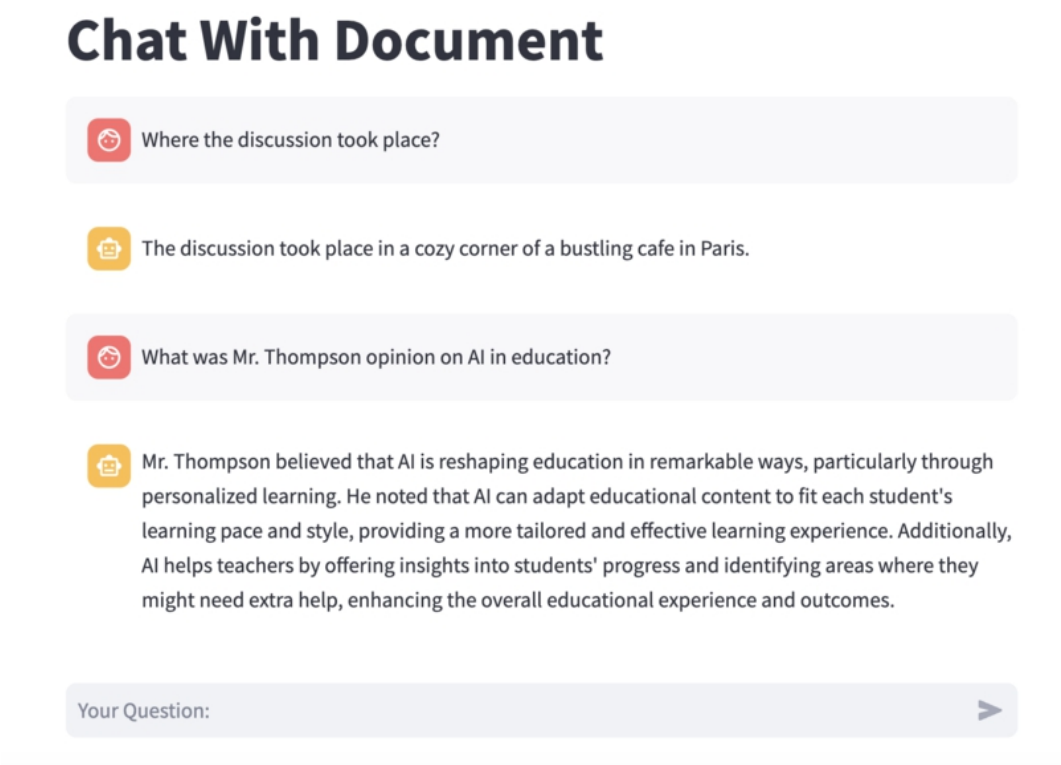

Figure 47. Controlling RAG Chain Response With pick()

Switching the LLM

If you want to change the LLM used for this application, you need to also change the embedding model used for the vector generation.

For Google Gemini, you can import the `GoogleGenerativeAIEmbeddings` model as follows:

```
from langchain_google_genai import ChatGoogleGenerativeAI,
GoogleGenerativeAIEmbeddings

# llm
GOOGLE_GEMINI_KEY = config("GOOGLE_GEMINI_KEY")
```

```
llm = ChatGoogleGenerativeAI(
    model="gemini-1.5-pro-latest", google_api_key=GOOGLE_GEMINI_KEY
)

# embeddings
embeddings = GoogleGenerativeAIEmbeddings(
    google_api_key=GOOGLE_GEMINI_KEY, model="models/embedding-001"
)
```

For Ollama, you can use the community-developed `OllamaEmbeddings` class like this:

```
from langchain_community.chat_models import ChatOllama
from langchain_community.embeddings import OllamaEmbeddings

# llm
llm = ChatOllama(model="mistral")

# embeddings
embeddings = OllamaEmbeddings(model="mistral")
```

Make sure that you use the same `model` when instantiating the `ChatOllama` and `OllamaEmbeddings` classes.

Summary

The code for this chapter is available in the folder *11_Chat_With_Document* from the book source code.

In this chapter, you've learned how to create a Chat With Document application using LangChain.

Using the RAG technique, LangChain can be used to retrieve information from a document, and then pass the information to the LLM.

The first thing you need to do is to process the document and turn it into chunks, which can then be converted into vectors using an embedding model.

The vectors are stored in ChromaDB, and the retriever created from the database is used when the user sends a query or input.

Once the core functionality is finished, you can easily add a web interface and chat memory into the application using Streamlit.

Next, let's see how we can load documents in different formats, such as `.docx` and `.pdf`.

CHAPTER 12: UPLOADING DIFFERENT DOCUMENT TYPES

Now that we can load and interact with a `.txt` document in LangChain, let's improve the application so we can also load other document types or formats, such as `.docx` and `.pdf`.

To load different document types, you need to create a file uploader interface in Streamlit.

Just below the Streamlit `title`, add new code as follows:

```
st.title("Ask the Document")

uploaded_file = st.file_uploader("Upload your document: ", type=["pdf", "docx",
"txt"])
add_file = st.button("Submit File", on_click=clear_history)

if uploaded_file and add_file:
    process_file(uploaded_file)
```

The `st.file_uploader()` method creates a file uploader input, then we add a button to submit the file.

When the submit button is clicked, the `clear_history()` method will be executed, so you need to define the method next:

```
def clear_history():
    if "langchain_messages" in st.session_state:
        del st.session_state["langchain_messages"]
```

Above the `clear_history()` function, create a new function named `process_file()` to process the uploaded file.

This function needs to check on the extension of the document to use the right loader:

```
def process_file(file):
    with st.spinner("File is in process..."):
        data = file.read()
        file_name = os.path.join("./", file.name)
        with open(file_name, "wb") as f:
            f.write(data)
        name, extension = os.path.splitext(file_name)

        if extension == ".pdf":
            from langchain_community.document_loaders import PyPDFLoader

            loader = PyPDFLoader(file_name)
        elif extension == ".docx":
            from langchain_community.document_loaders import Docx2txtLoader

            loader = Docx2txtLoader(file_name)
        elif extension == ".txt":
            from langchain_community.document_loaders import TextLoader

            loader = TextLoader(file_name)
        else:
            st.error("This format is not supported!")
```

The LangChain loaders simply load the respective document using Python packages.

You need to install the `PyPDF` and `Docx2txt` packages used by the loaders:

```
pip install pypdf == 4.2.0 docx2txt == 0.8
```

Once the `loader` is created, you can load and split the document into chunks, convert the chunks into vectors, and create the chains as before:

```
    docs = loader.load()
      text_splitter = RecursiveCharacterTextSplitter(
          chunk_size=1000, chunk_overlap=200
      )
      chunks = text_splitter.split_documents(docs)
      embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_KEY)
      vector_store = Chroma.from_documents(chunks, embeddings)
      retriever = vector_store.as_retriever()
      history_aware_retriever = create_history_aware_retriever(
          llm, retriever, contextualize_prompt
      )
      rag_chain = create_retrieval_chain(
          history_aware_retriever, question_answer_chain
      )
      conversational_rag_chain = RunnableWithMessageHistory(
          rag_chain,
          lambda session_id: history,
          input_messages_key="input",
          history_messages_key="chat_history",
          output_messages_key="answer",
      )
```

To make the `conversational_rag_chain`, available outside of the function, we need to persist the object by saving it in Streamlit session state.

We also show a message saying the file is uploaded and embedded successfully:

```
    st.session_state.crc = conversational_rag_chain
      st.success("File uploaded and embedded successfully")
```

When the user passes a question, you can check if `crc` is available in `st.session_state` and use it.

Also, don't forget to call the `pick()` method from the `crc` chain to stream the answer:

```
if question:
    with st.chat_message("user"):
        st.markdown(question)
    if "crc" in st.session_state:
        crc = st.session_state["crc"]
        answer_chain = crc.pick("answer")
        response = answer_chain.stream(
            {"input": question}, config={"configurable": {"session_id": "any"}}
        )
        with st.chat_message("assistant"):
            st.write_stream(response)
    else:
        st.error("No document is uploaded. Upload your document first")
```

Here, we also add an `else` statement to inform the user when no document has been uploaded yet.

And that's it. Now you can upload a `.txt`, `.docx`, or `.pdf` document and ask questions relevant to the content of the document.

If you upload a document of other types, Python will show 'The document format is not supported' error. Good job!

Summary

The code for this chapter is available in the folder *12_Uploading_Different_Document_Types* from the book source code.

In this chapter, you have improved the Chat With Document application further by creating an interface for uploading a file, then load the document using different loaders, depending on the type of the document.

In the next chapter, I’m going to show you how to chat with YouTube videos. See you there!

CHAPTER 13: CHAT WITH YOUTUBE VIDEOS

Now that you've learned how to make AI interact with documents, let's continue with creating a Chat with YouTube application.

To create this application, you can copy the finished application from Chapter 12 and make the changes shown in this chapter.

Let's get started!

Adding The YouTube Loader

The Chat With YouTube application uses the RAG technique to augment the LLM with the video's transcript data.

To get a YouTube video's transcript, you need to install the `youtube-transcript-api` package using pip as follows:

```
pip install youtube-transcript-api==0.6.2
```

This API is used by LangChain's YouTube loader to fetch the transcript of a video.

At the top of the file, import the `YoutubeLoader` class from LangChain like this:

```
from langchain_community.document_loaders import YoutubeLoader
```

Just below the Streamlit title, create a Streamlit text input to replace the file uploader as follows:

```
youtube_url = st.text_input("Input YouTube URL")
if youtube_url:
    process_youtube_url(youtube_url)
```

Next, create the `process_youtube_url()` function which replaces the `process_file()` function.

Instead of importing different loaders to load the documents, you can use the `YoutubeLoader` to create a document from the YouTube `url` as follows:

```
def process_youtube_url(url):
    with st.spinner("Processing YouTube video..."):
        loader = YoutubeLoader.from_youtube_url(url)

        docs = loader.load()
        text_splitter = RecursiveCharacterTextSplitter(
            chunk_size=1000, chunk_overlap=200
        )
        #... the rest of the function
        #...
        st.success("Video processed. Ask your questions")
```

The `YoutubeLoader.from_youtube_url()` method will fetch a YouTube video's transcript as a document, which you can then split into chunks using the `text_splitter` object.

The rest of the function code is the same, except that when the `rag_chain` is created, we show the 'video processed' success

message.

Now run the application using `streamlit run`, pass the URL, and ask questions relevant to the YouTube video's content.

For example, I passed the URL to my video at https://youtu.be/Sr4KeW078P4 which explains the JavaScript Promise syntax:

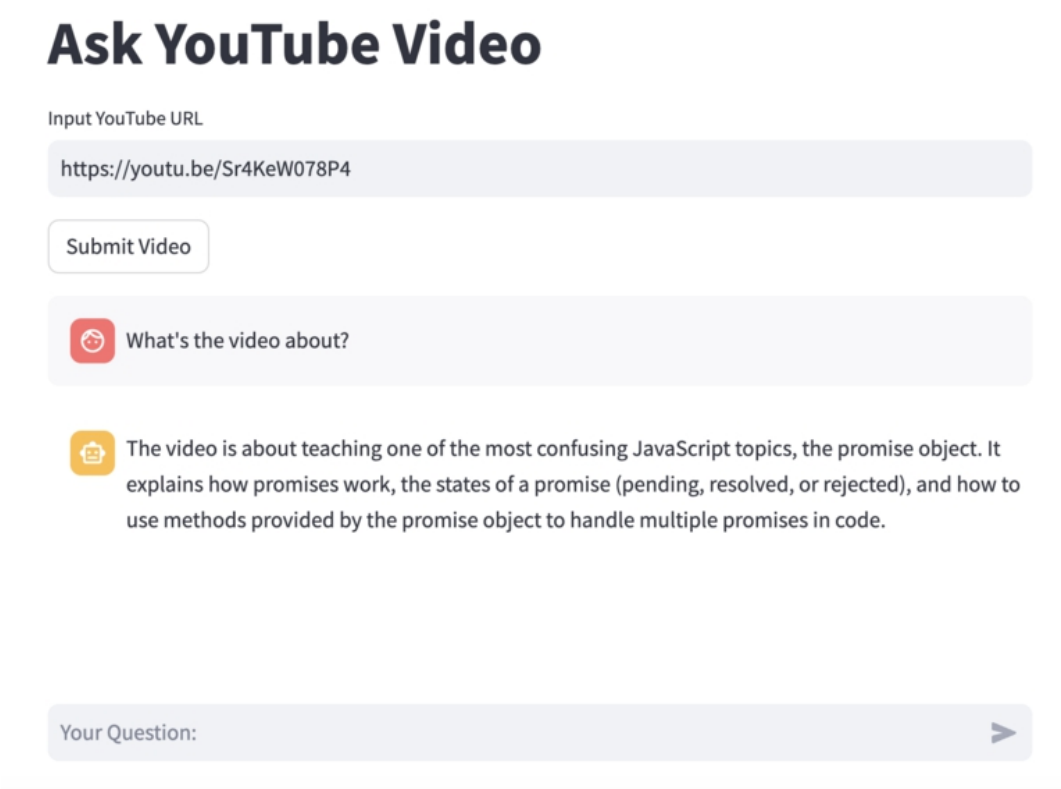

Figure 48. Ask YouTube Application Result

You can pass a YouTube short link or full link. It will work as shown above.

Handling Transcript Doesn't Exist Error

Because YouTube music videos don't have transcripts, you'll get an error when you give one to the application:

```
ValueError: Expected IDs to be a non-empty list, got []
```

This error occurs because the `YoutubeLoader` loads an empty document list, so that empty list gets passed into `Chroma.from_documents()` method.

To prevent this error, you need to check if the `docs` variable is not empty before running further process:

```
if docs:
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=1000, chunk_overlap=200
    )
    chunks = text_splitter.split_documents(docs)
    # ...
    st.session_state.crc = conversational_rag_chain
    st.success("Video processed. Ask your questions")
else:
    st.error("Video has no transcript. Please try another video")
```

This way, an error message is shown when the video has no transcript, and no error is thrown by ChromaDB:

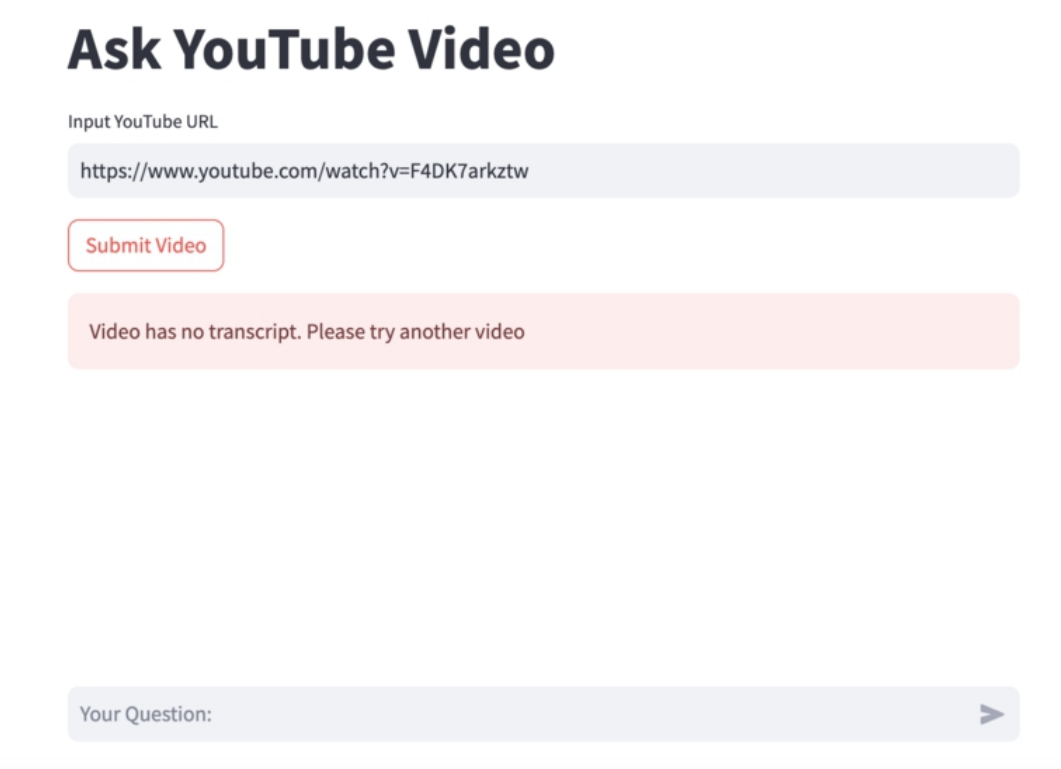

Figure 49. No Transcript Detected

Without the transcript, we can't generate vector data.

Summary

The code for this chapter is available in the folder *13_Chat_With_Youtube* from the book source code.

In this chapter, you've learned how to create a Chat With YouTube application that fetches a YouTube video transcript using the URL, and then converts the transcript into vectors, which can be used to augment the LLM knowledge.

By using AI, you can ask for the key points or summary of a long video without having to watch the video yourself.

CHAPTER 14: INTERACTING WITH IMAGES USING MULTIMODAL MESSAGES

Now that we can load different types of documents, let's continue with making AI understand images.

Understanding Multimodal Messages

A multimodal message is a message that uses more than one mode of communication to communicate.

The communication modes known to humans are:

- Text
- Video
- Audio

Image

Advanced models like GPT-4 and Gemini Pro already have vision capabilities, meaning the models can "see" images and answer questions about them.

By sending a multimodal message, we can use AI to interact with images, like asking how many persons are shown on an image or what color is dominant in the image.

Let’s learn how to create such application using LangChain next.

Sending Multimodal Messages in LangChain

To send a multimodal message in LangChain, you only need to adjust the prompt template and pass a list instead of a string for the human message.

First, create a new file named `handle_image.py`, import the required packages, and define the LLM to use as follows:

```
from decouple import config
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

import base64

OPENAI_KEY = config("OPENAI_KEY")

llm = ChatOpenAI(model="gpt-4o", api_key=OPENAI_KEY)
```

The base64 module will be used to encode images in base 64 format.

Write the function to encode images below the `llm` variable:

```
def encode_image(image_path):
    with open(image_path, "rb") as image_file:
        return base64.b64encode(image_file.read()).decode("utf-8")


image = encode_image("./image-1.jpg")
```

You can pass any image you have on your computer to the `encode_image()` function. Just make sure it doesn't contain any sensitive data.

The encoded image can then be passed into a chat prompt template as follows:

```
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a helpful assistant that can describe images in
detail."),
        (
            "human",
            [
                {"type": "text", "text": "{input}"},
                {
                    "type": "image_url",
                    "image_url": {
                        "url": f"{image}",
                        "detail": "low",
                    },
                },
            ],
        ),
    ]
)
```

In the template above, you can see that there are two messages passed as the human message: the "text" for the question and

the "image_url" for the image.

We can pass a public image url such as 'https://', but here we use the `data:image` URI because we upload an image from our computer.

Behind the scenes, LangChain automatically converts the message into a multimodal message.

Create a chain from the `prompt` and the `llm`, and ask the user for an input:

```
chain = prompt | llm
response = chain.invoke({"input": "What do you see on this image?"})

print(response.content)
```

The LLM will process the image, and then give you the appropriate answer.

Adding UI and Chat History

Now that you can ask questions about an image using a language model, let's improve the application by adding a web interface and chat history.

As always, import the packages needed to add the web interface and history first:

```
from decouple import config
from langchain_openai import ChatOpenAI
import streamlit as st

# Adding History
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_community.chat_message_histories import
StreamlitChatMessageHistory
```

```
from langchain_core.runnables.history import RunnableWithMessageHistory

import os, base64
```

Next, you need to instantiate Streamlit chat history and upgrade the chain to include chat history, similar to Chapter 9.

You also need to create a `process_image()` function, which will encode the image and store it in session state:

```
history = StreamlitChatMessageHistory()

chain = prompt | llm

chain_with_history = RunnableWithMessageHistory(
    chain,
    lambda session_id: history,
    input_messages_key="input",
    history_messages_key="chat_history",
)

def process_image(file):
    with st.spinner("Processing image..."):
        data = file.read()
        file_name = os.path.join("./", file.name)
        with open(file_name, "wb") as f:
            f.write(data)
        image = encode_image(file_name)
        st.session_state.encoded_image = image
        st.success("Image encoded. Ask your questions")
```

The next step is to create a function to clear chat history, then create a file uploader using Streamlit.

When an image is uploaded, process that image using the `process_image()` function:

```
def clear_history():
    if "langchain_messages" in st.session_state:
        del st.session_state["langchain_messages"]
```

```
st.title("Chat With Image")

uploaded_file = st.file_uploader("Upload your image: ", type=["jpg", "png"])
add_file = st.button("Submit Image", on_click=clear_history)

if uploaded_file and add_file:
    process_image(uploaded_file)
```

With the image encoded and saved in session state, you only need to display chat history, create a chat input, and run the `chain_with_history.stream()` method to stream the answer.

This part is similar to previous applications:

```
for message in st.session_state["langchain_messages"]:
    role = "user" if message.type == "human" else "assistant"
    with st.chat_message(role):
        st.markdown(message.content)

question = st.chat_input("Your Question")
if question:
    with st.chat_message("user"):
        st.markdown(question)
    if "encoded_image" in st.session_state:
        image = st.session_state["encoded_image"]
        response = chain_with_history.stream(
            {"input": question, "image": image},
            config={"configurable": {"session_id": "any"}},
        )
        with st.chat_message("assistant"):
            st.write_stream(response)
    else:
        st.error("No image is uploaded. Upload your image first.")
```

When there's no image, display an error message saying "No image is uploaded. Upload your image first."

Now run the application using the `streamlit run` command, and you can ask questions about the image you uploaded.

I'm using an image from https://g.codewithnathan.com/lc-image for the example below:

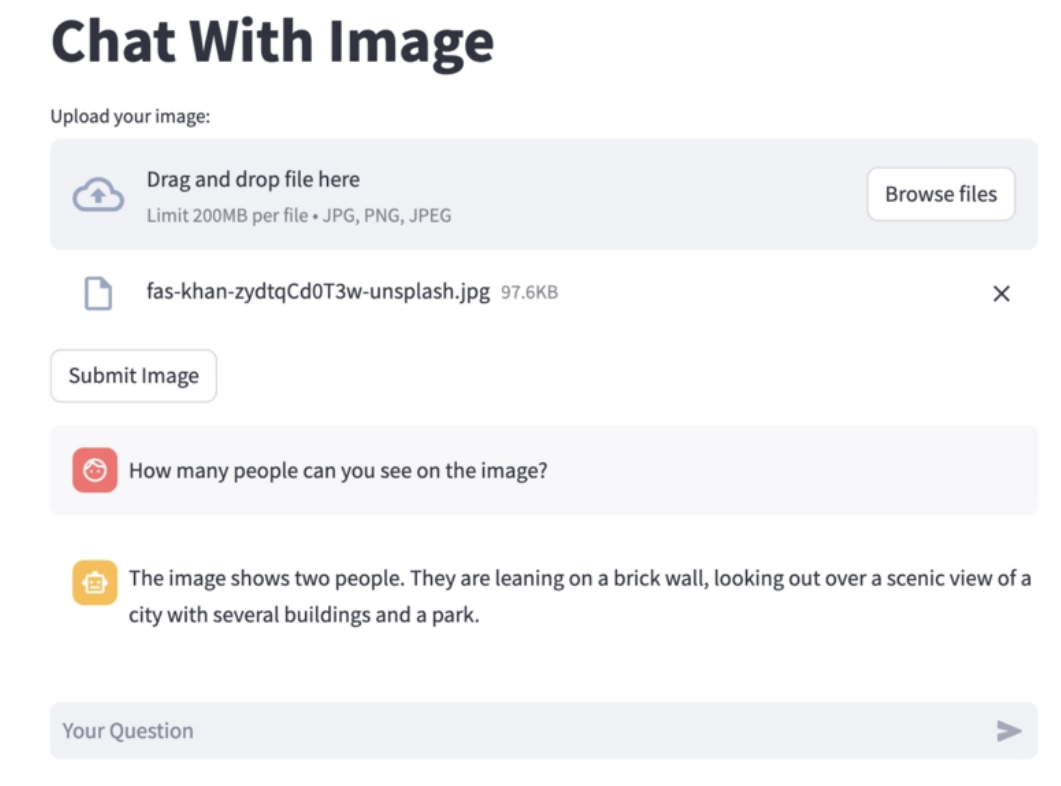

Figure 50. Chat With Image Result

Try asking for a certain detail, such as how many people can you see on the image, or what color is dominant on the image.

Ollama Multimodal Message

If you want to send a multimodal message to Ollama, you need to download a model that supports the message format, such as `bakllava` and `llava`.

You can run the command `ollama pull bakllava` to download the model to your machine. Note that both models require 8GB of RAM to run without any issue.

To send a multimodal message to Ollama, you need to make sure that the `image_url` key holds a string value as follows:

```
{
    "type": "image_url",
    "image_url": f"""{image}",
},
```

But then again, there's a bug on LangChain side that converts the `image_url` key to a dictionary and add a `url` key in that dictionary.

I have reported this issue to LangChain maintainers at https://github.com/langchain-ai/langchain/issues/22460

You need to open the `ollama.py` source code and change the code at line 123 as follows:

```
if isinstance(content_part.get("image_url")["url"], str):
    image_url_components = content_part["image_url"]["url"].split(",")
```

I will update this section once the issue is resolved.

Summary

The code for this chapter is available in the folder *14_Handling_Images* from the book source code.

In this chapter, you've learned how to send a multimodal message to a language model using LangChain.

Note that not all models can understand a multimodal message. If you're using Ollama, you need to use a model like `bakllava` and not `mistral` or `gemma`.

If the LLM doesn’t understand, it will usually tell you that it can’t understand the message you’re sending.

CHAPTER 15: DEPLOYING AI APPLICATION TO PRODUCTION

Now that you've created various AI-powered applications, it's time to learn how to deploy one to the internet so others can use it.

In this chapter, we will learn how to deploy applications that use Google Gemini and OpenAI GPT.

Note that we won't explore how to deploy an application that's powered by Ollama because we need to self-host Ollama to run the application properly.

Because we created our applications using Streamlit, one way to deploy the application is to launch it on the Streamlit Cloud Community.

Keep in mind that because the Streamlit Cloud Community is free, your application might be put to sleep when inactive to save server resources.

Creating a Sidebar in Streamlit

There's a little change we need to make in our application before deploying it.

Notice that the applications you created are all using your own API key to access the LLM provider's API.

This is not recommended for production because you will be charged every time a user uses your application.

Instead of supplying your API key for use, you need to enable users to add their own API keys.

This is easy in Streamlit because you only need to create a text input in the sidebar for the API key.

In our application, create a sidebar before instantiating the `llm` object as follows:

```
st.title("Ask YouTube Video")

with st.sidebar:
    st.title("Add Your API Key First")
    openai_key = st.text_input("OpenAI API Key", type="password")
if not openai_key:
    st.info("Enter your OpenAI API Key to continue")
    st.stop()

llm = ChatOpenAI(model="gpt-4o", api_key=openai_key)
```

In the code above, we created a sidebar using the `with st.sidebar` syntax, then added the title and text input elements to the sidebar.

When the `openai_key` variable is empty, we run the `st.info()` method to display an instruction to the user, then call the

`st.stop()` method so that Streamlit stops further execution of your Python code.

This way, the rest of the interface won't show until an API key is added:

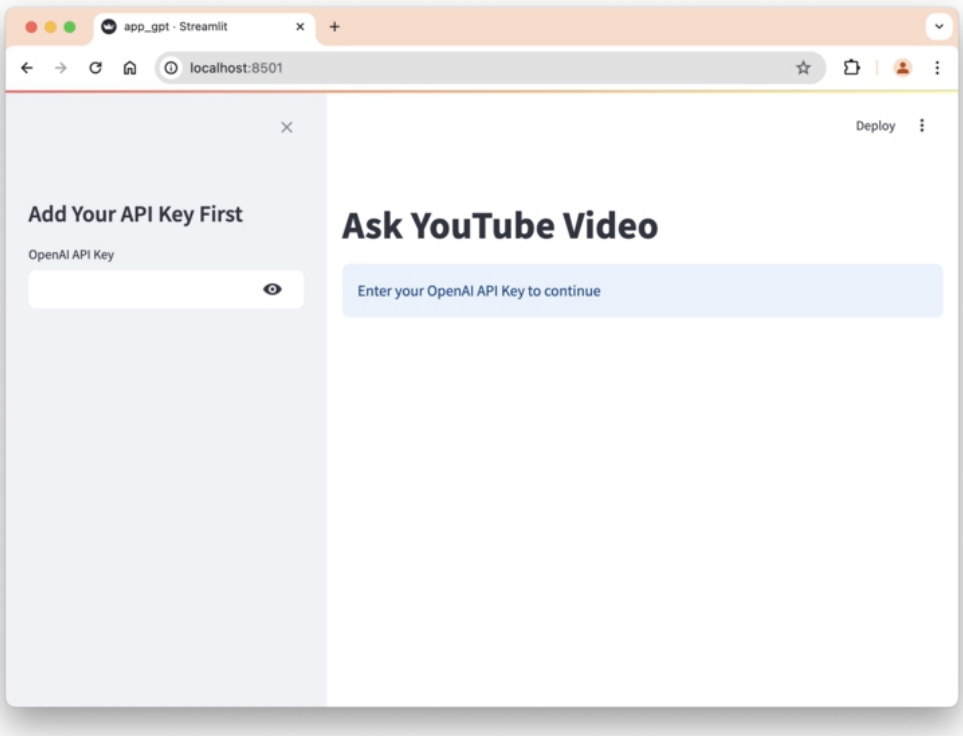

Figure 51. Streamlit Asking for API Key

Once you added the API key, the rest of the interface will be shown:

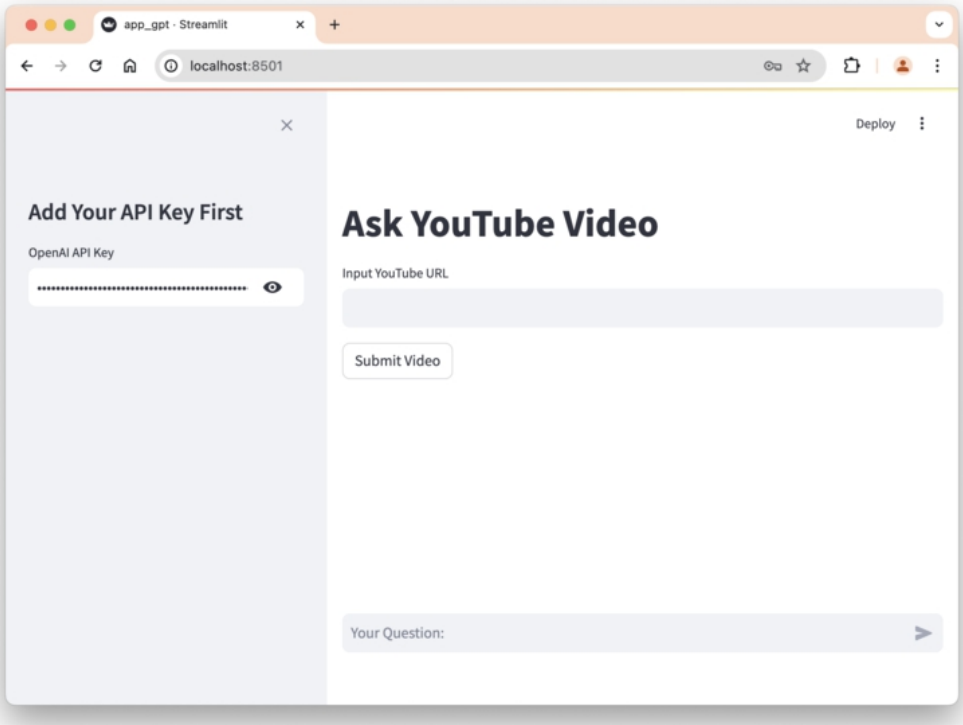

Figure 52. Streamlit UI Shown When API Key is Added

Alright, now you can remove the `from decouple import config` line because it's not needed anymore.

Adding requirements.txt file in the Project Folder

The next step is to include a `requirements.txt` file in the project folder.

This file will be used by Streamlit to install the dependencies needed to run the application.

Here's the content of the `requirements.txt` file:

```
langchain == 0.2.1
langchain_community==0.2.1
```

```
langchain_core==0.2.3
langchain-openai==0.1.7
streamlit==1.35.0
chromadb==0.4.18
youtube-transcript-api==0.6.2
```

Notice that the `langchain_community` and `langchain_core` packages are also included in the `.txt` file above.

Streamlit Cloud uses `uv` instead of `pip` to install packages, and `uv` doesn't automatically install the extra packages when you install `langchain`, so you need to specify them.

Creating a GitHub Repository

Deploying a Streamlit application requires you to upload your code to GitHub. GitHub is a platform that you can use to host and share your software project.

Head over to https://github.com and login or register for a new account if you don't have one already.

From the dashboard, create a new repository by clicking + New on the left sidebar, or the + sign on the right side of the navigation bar:

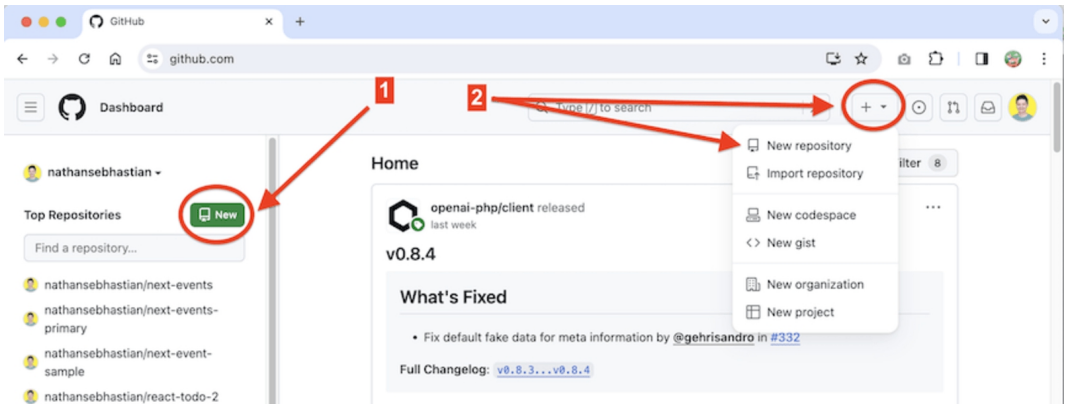

Figure 53. Two Ways To Create a Repository in GitHub

A repository (or repo) is a storage space used to store software project files.

In the *Create a Repository* page, fill in the details of your project. The only required detail is the repository name:

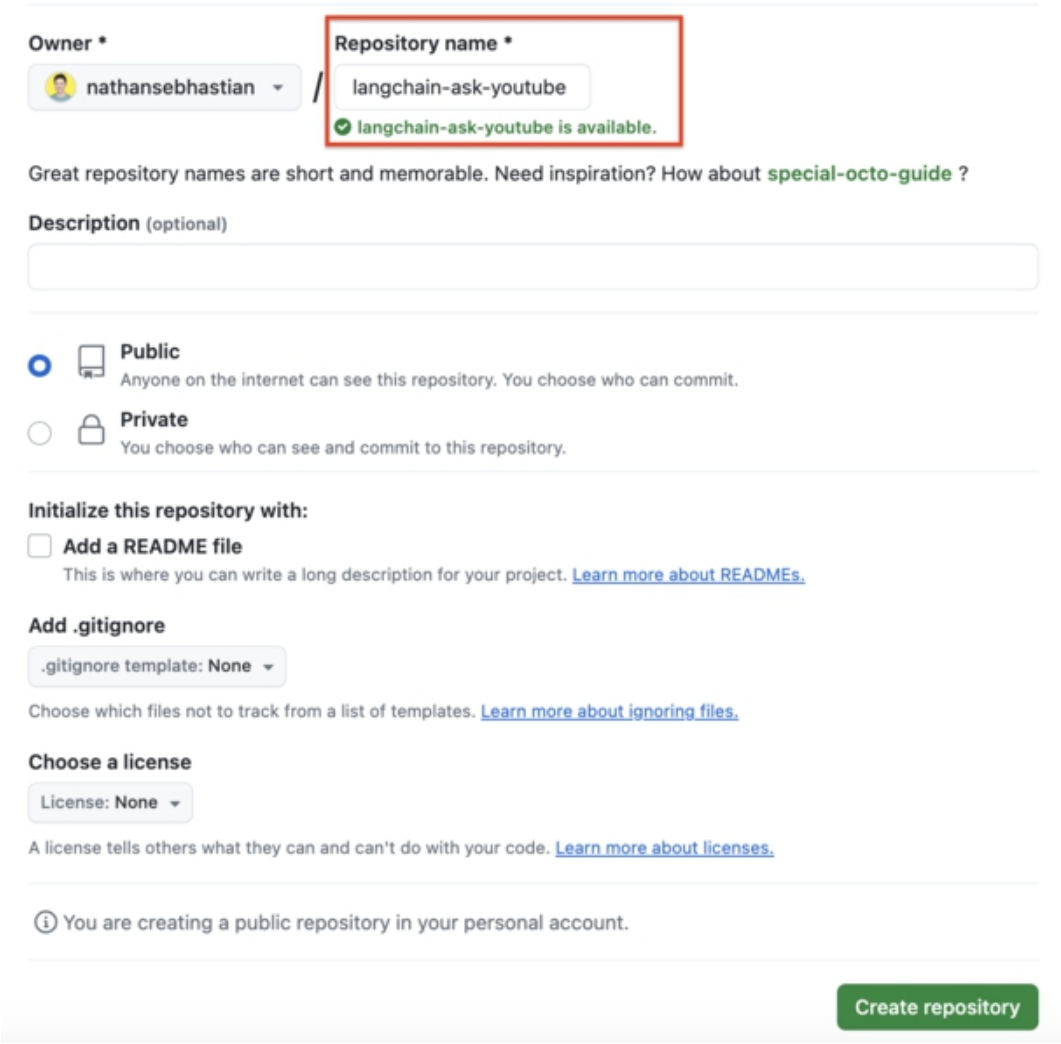

Figure 54. Create a GitHub Repository

You can make the repository public if you want this project as a part of your portfolio, or you can make it private.

Once a new repo is created, you will be given instructions on how to push your files into the repository.

You need to follow the instructions for pushing an existing repo:

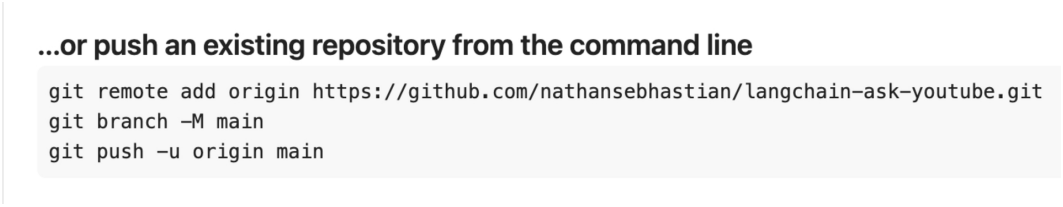

Figure 55. Git Push Instructions

Now you need to create a repository for your project. Open the command line on the project folder, then run the `git init` command:

```
git init
```

This will turn your project into a local repository. Add all project files into this local repo by running the `git add .` command:

```
git add .
```

Changes added to the repo aren't permanent until you run the `git commit` command. Commit the changes as shown below:

```
git commit -m 'Ready for Deployment'
```

The `-m` option is used to add a message for the commit. Usually, you summarize the changes committed to the repository as the message.

Now you need to push this existing repository to GitHub. You can do so by following the GitHub instructions:

```
git remote add origin <URL>
git branch -M main
git push -u origin main
```

You might be asked to enter your GitHub username and password when running the `git push` command.

Once the push is complete, refresh the GitHub repo page on the browser, and you should see your project files and folders there.

This means our application is already pushed (uploaded) to a remote repository hosted on GitHub.

The remote server where we're going to deploy our application can fetch the code from this repository.

You can check my GitHub repo at https://github.com/nathansebhastian/langchain-ask-youtube

Deploying to Streamlit

The application is now ready for deployment. There are two ways you can deploy to Streamlit:

1. From Streamlit website at https://share.streamlit.io
2. From the localhost with the `streamlit run` command

I'm going to show you the first way, then continue with the second way.

You need to access Streamlit's Community Cloud at https://share.streamlit.io and register for an account:

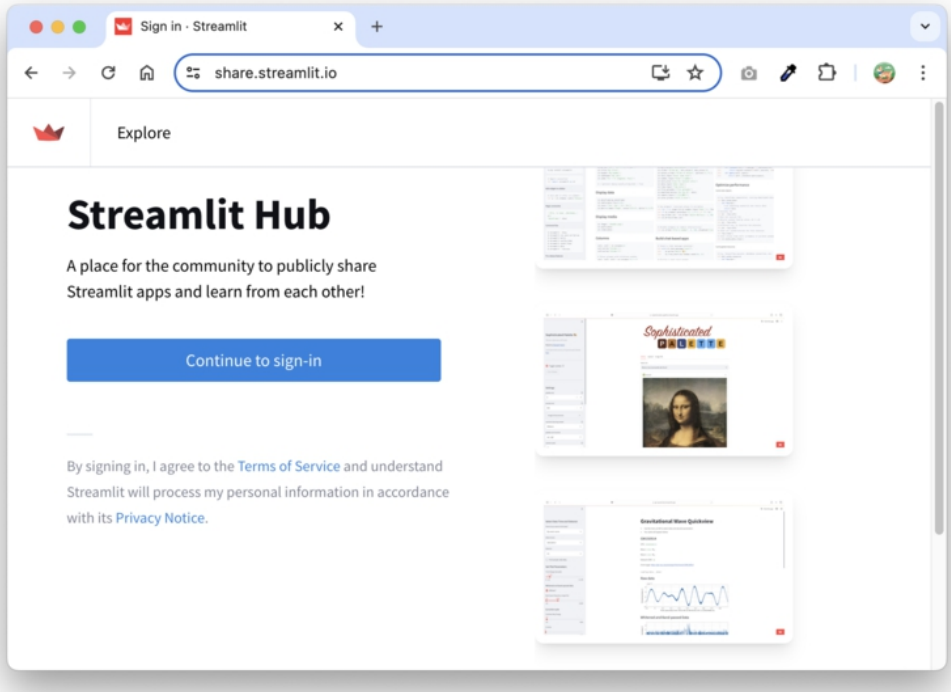

Figure 56. Streamlit Share Homepage

After you create an account, you will be directed to a workspace. Click the 'Create app' link at the top-right corner:

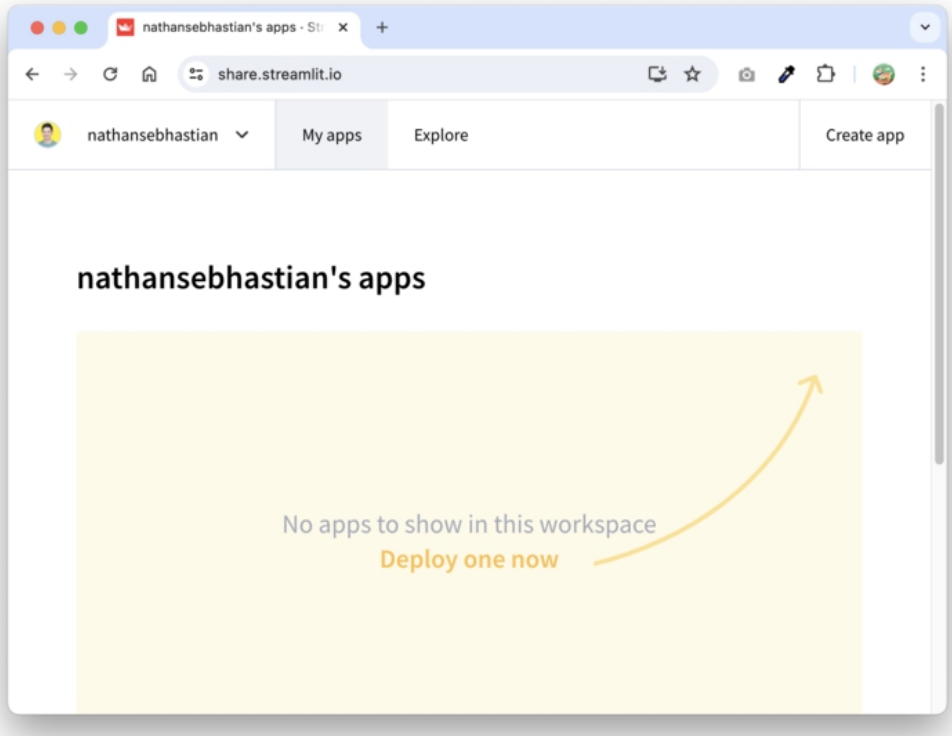

Figure 57. Streamlit 'Create app' Link

Streamlit will then ask to connect to Github. Click the 'Connect to GitHub' button as shown below:

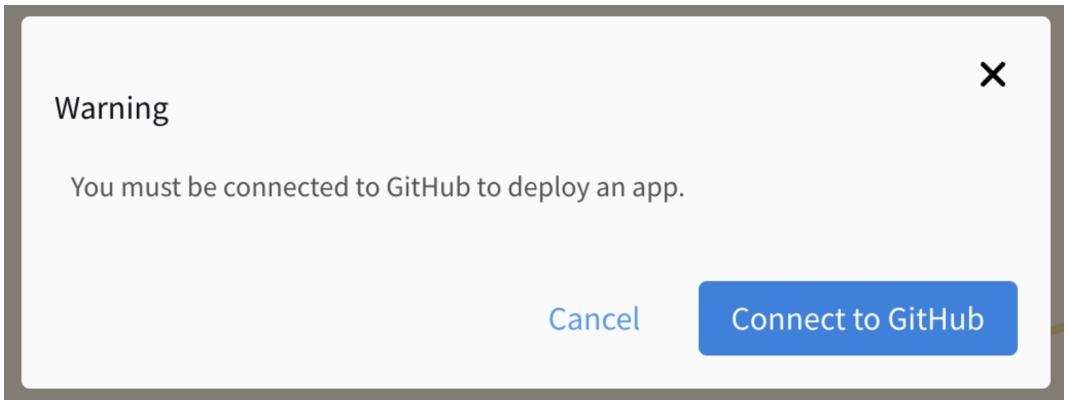

Figure 58. Streamlit Connect to GitHub

Then authorize Streamlit to access your GitHub repository.

On the next screen, select 'I have an app' as shown below:

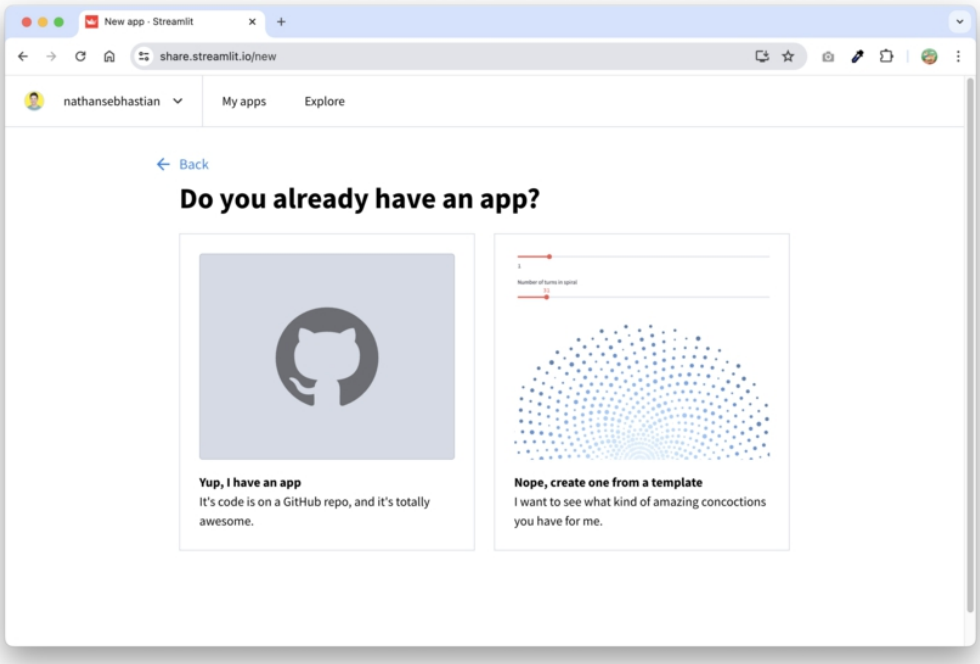

Figure 59. Streamlit Deploy Existing Application

Next, you need to fill in the details of your application on the Deploy page.

Add the code repository you've created in the previous section, then set the Branch, Main file path, and App URL.

You can customize the App URL to make it shorter. Use my input below as a reference:

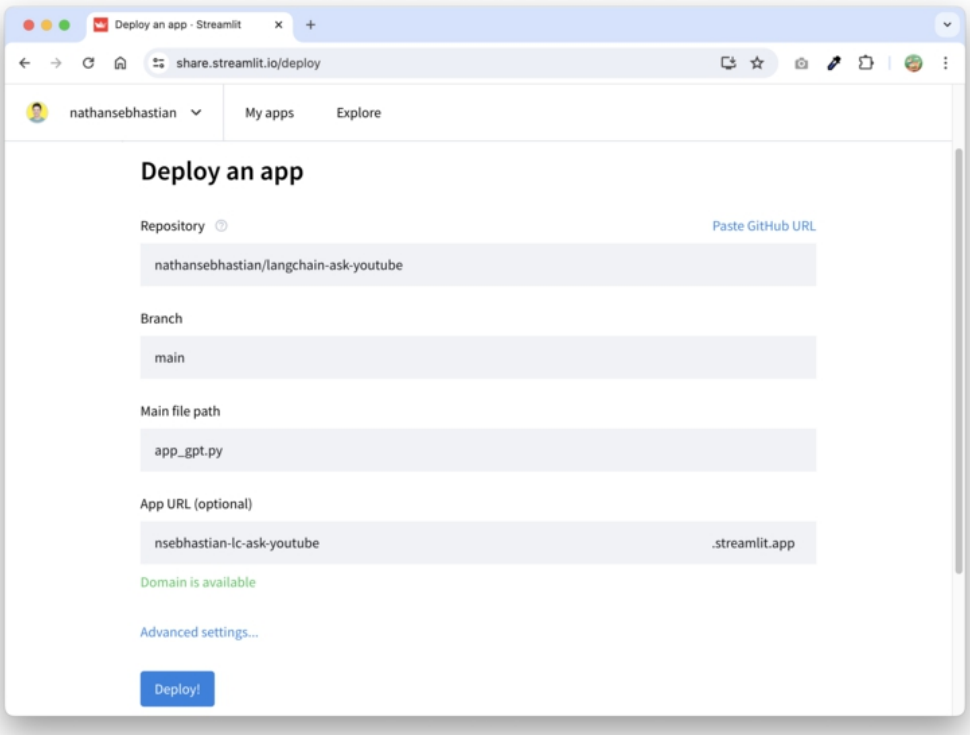

Figure 60. Streamlit Deploy Page

Once you're finished, click 'Deploy' so that Streamlit will fetch the code from GitHub and deploy it to their community cloud server.

Once done, you can access the application from a generated `streamlit.app` URL.

You can check my deployment at https://nsebhastian-lc-ask-youtube.streamlit.app

The second way you can deploy to the Streamlit Cloud is by running the `streamlit run` command to run your application on localhost, then click the 'Deploy' link at the top-right corner:

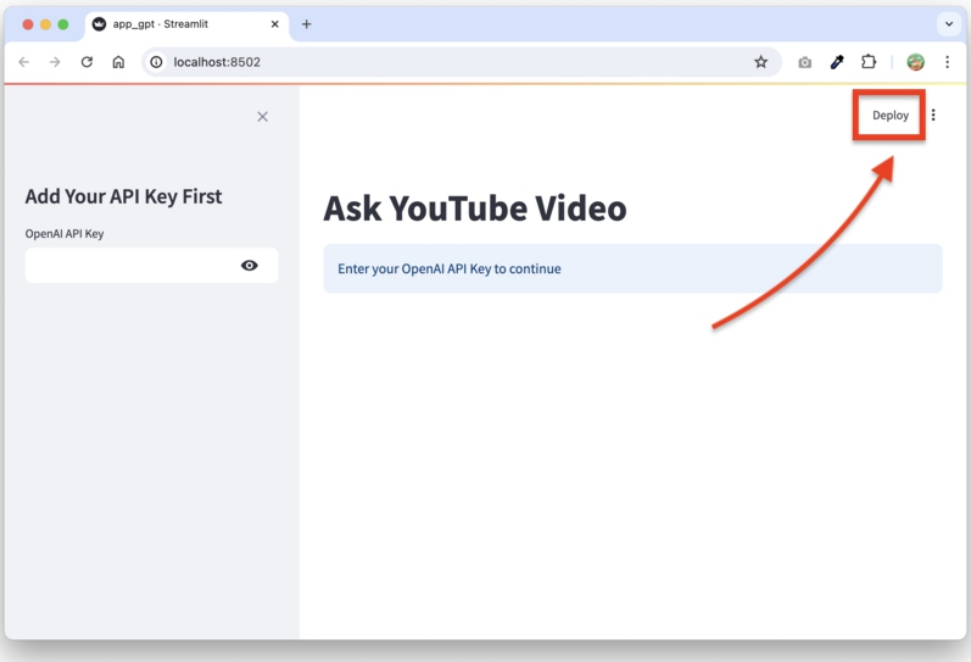

Figure 61. Streamlit Deploy From Localhost

From there, select Streamlit Cloud Community to deploy for free:

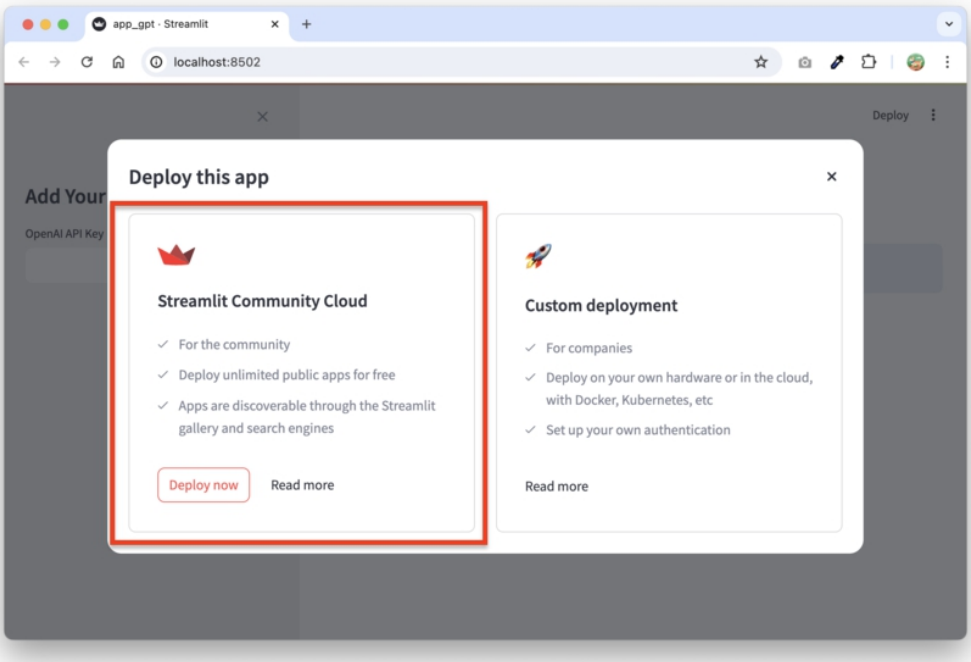

Figure 62. Streamlit Select Cloud Community

You will be taken to the same Deploy page, provided that you have uploaded the project to a GitHub repository. Click the 'Deploy' button once you're done.

Summary

The code changes in this chapter can be found at https://github.com/nathansebhastian/langchain-ask-youtube which I linked before.

You have successfully deployed a Streamlit application to the internet. Now you can share your hard work with everyone. Nice work!

As you can see, deploying an AI-powered application is not so different than deploying a regular web application.

Because AI models are accessible using HTTP protocol, you only need to ensure that an API key to access the models is added to the application.

Instead of providing your own key and incurring charges from running the model, you can ask the users to provide their own keys for their own use.

WRAPPING UP

Congratulations on finishing this book! We've gone through many concepts and topics together to help you learn how to develop an AI-powered application using LangChain, Streamlit, GPT, Gemini, and Ollama.

You've also learned how to deploy the application to Streamlit Cloud Community so that it can be accessed from the internet.

I hope you enjoyed learning and exploring LangChain with this book as much as I enjoyed writing it.

I'd like to ask you for a small favor.

If you enjoyed the book, I'd be very grateful if you would leave an honest review on Amazon (I read all reviews coming my way)

Every single review counts, and your support makes a big difference.

Thanks again for your kind support!

Until next time,

Nathan

ABOUT THE AUTHOR

Nathan Sebhastian is a senior software developer with 8+ years of experience in developing web and mobile applications.

He is passionate about making technology education accessible for everyone and has taught online since 2018.

LangChain Programming For Beginners

A Step-By-Step Guide to AI Application Development With LangChain, Python, OpenAI/ChatGPT, Google/Gemini and Other LLMs

By Nathan Sebhastian

https://codewithnathan.com