

6

Building Conversational Applications

With this chapter, we embark on the hands-on section of this book, with our first concrete implementation of LLM-powered applications. Throughout this chapter, we will cover a step-by-step implementation of a conversational application, using LangChain and its components, building on the knowledge you've gained from the previous chapters. By the end of this chapter, you will be able to set up your own conversational application project with just a few lines of code.

We will cover the following key topics:

- Configuring the schema of a simple chatbot
- Adding the memory component
- Adding non-parametric knowledge
- Adding tools and making the chatbot “agentic”
- Developing the front-end with Streamlit

Technical requirements

To complete the tasks in this chapter, you will need the following:

- A Hugging Face account and user access token.
- An OpenAI account and user access token.
- Python 3.7.1 or a later version.
- Python packages – make sure to have the following Python packages installed: `langchain`, `python-dotenv`, `huggingface_hub`, `streamlit`, `openai`, `pypdf`, `tiktoken`, `faiss-cpu`, and `google-search-results`. They can be easily installed via `pip install` in your terminal.

You'll find the code for this chapter in the book's GitHub repository at <https://github.com/PacktPublishing/Building-LLM-Powered-Applications>.

Getting started with conversational applications

A conversational application is a type of software that can interact with users using natural language. It can be used for various purposes, such as providing information, assistance, entertainment, or transactions. Generally speaking, a conversational application can use different modes of communication, such as text, voice, graphics, or even touch. A conversational application can also use different platforms, such as messaging apps, websites, mobile devices, or smart speakers.

Today, conversational applications are being taken to the next level thanks to LLMs. Let's look at some of the benefits that they provide:

- Not only do LLMs provide a new level of natural language interactions, but they can also enable applications to perform reasoning

based on the best responses, given users' preferences.

- As we saw in previous chapters, LLMs can leverage their parametric knowledge, but are also enriched with non-parametric knowledge, thanks to embeddings and plug-ins.
- Finally, LLMs are also able to keep track of the conversation thanks to different types of memory.

The following image shows what the architecture of a conversational bot might look like:

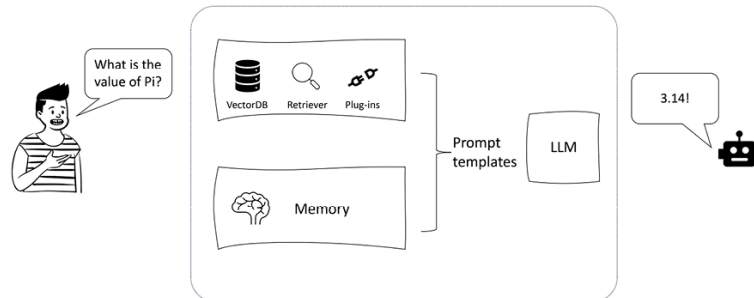


Figure 6.1: Sample architecture of a conversational bot

Throughout this chapter, we will build from scratch a text conversational application that is able to help users plan their vacations. We will call this app GlobeBotter. We will add incremental layers of complexity to make the app as enjoyable as possible for the end user.

So, let's start with the basics behind a conversational app architecture.

Creating a plain vanilla bot

To start with, let's initialize our LLM and set the schema for our bot. The schema refers to the type of messages the bot is able to receive. In our case, we will have three types of messages:

- **System message:** The instructions we give the bot so that it behaves as a travel assistant.
- **AI Message:** The message generated by the LLM
- **Human Message:** The user's query

Let's start with a simple configuration:

```
from langchain.schema import (
    AIMessage,
    HumanMessage,
    SystemMessage
)
from langchain.chains import LLMChain, ConversationChain
from langchain.chat_models import ChatOpenAI
from langchain.chat_models import ChatOpenAI
chat = ChatOpenAI()
messages = [
    SystemMessage(content="You are a helpful assistant that help the user to plan an optim
    HumanMessage(content="I'm going to Rome for 2 days, what can I visit?")]
```

We can then save and print the output as follows:

```
output = chat(messages)
print(output.content)
```

Here is the output:

```
In Rome, there are many famous attractions to visit. Here's an optimized itinerary for you
Day 1:
1. Start your day by visiting the Colosseum, one of the world's most iconic ancient landm
2. Next, explore the nearby Roman Forum, an ancient Roman marketplace.
3. Afterward, head to the Pantheon, a well-preserved Roman temple with a stunning dome.
4. Take a stroll through the historic district of Trastevere, known for its charming stree
5. In the evening, visit the Trevi Fountain and toss a coin to ensure your return to Rome.
Day 2:
1. Begin your day at Vatican City, the smallest independent state in the world. Visit St.
2. Explore the Vatican Museums, home to an extensive collection of art and historical arti
3. Enjoy a leisurely walk along the Tiber River and cross over to the picturesque neighbor
4. Visit the Spanish Steps, a popular meeting point with a beautiful view of the city.
5. End your day by exploring the charming neighborhood of Piazza Navona, known for its bar
Remember to check the opening hours and availability of tickets for the attractions in adv
```

As you can see, the model was pretty good at generating an itinerary in Rome with only one piece of information from our side, the number of days.

However, we might want to keep interacting with the bot, so that we can further optimize the itinerary, providing more information about our preferences and habits. To achieve that, we need to add memory to our bot.

Adding memory

As we're creating a conversational bot with relatively short messages, in this scenario, a `ConversationBufferMemory` could be suitable. To make the configuration easier, let's also initialize a `ConversationChain` to combine the LLM and the memory components.

Let's first initialize our memory and chain (I'm keeping `verbose = True` so that you can see the bot keeping track of previous messages):

```
from langchain.memory import ConversationBufferMemory
from langchain.chains import ConversationChain
memory = ConversationBufferMemory()
conversation = ConversationChain(
    llm=chat, verbose=True, memory=memory
)
```

Great, now let's have some interactions with our bot:

```
conversation.run("Hi there!")
```

The following is the output:

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative an
Current conversation:
```

```
Human: Hi there!
AI:
> Finished chain.
'Hello! How can I assist you today?'
```

Next, we provide the following input:

```
conversation.run("what is the most iconic place in Rome?")
```

Here is the corresponding output:

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative and
Current conversation:
Human: Hi there!
AI: Hello! How can I assist you today?
Human: what is the most iconic place in Rome?
AI:
> Finished chain.
'The most iconic place in Rome is probably the Colosseum. It is a magnificent amphitheater'
```

As you can see from the chain, it is keeping track of the previous interactions. Let's challenge it and ask something related to the previous context:

```
conversation.run("What kind of other events?")
```

The following is the output that we receive:

```
> Entering new ConversationChain chain...
Prompt after formatting:
The following is a friendly conversation between a human and an AI. The AI is talkative and
Current conversation:
Human: Hi there!
AI: Hello! How can I assist you today?
Human: what is the most iconic place in Rome?
AI: The most iconic place in Rome is probably the Colosseum. It is a magnificent amphitheater
Human: What kind of other events?
AI:
> Finished chain.
'Other events that took place at the Colosseum include mock sea battles, animal hunts, and'
```

The bot was able to understand that our request was related to its previous answer. We can also retrieve the message history with the `memory.load_memory_variables()` method (you can see the full output in the GitHub repository). Here is a snippet of the output:

```
{'history': 'Human: Hi there!\nAI: Hello! How can I assist you today?\nHuman: what is the'}
```

Rather than running the `conversation.run` method at every interaction, I've coded a `while` cycle to make it interactive. The following is a snapshot of the whole conversation (you can find it in the book's GitHub repository):

```
while True:
    query = input('you: ')
    if query == 'q':
        break
    output = conversation({"input": query})
    print('User: ', query)
    print('AI system: ', output['response'])
```

The following is a truncated sample from the output (you can find the whole output in the book's GitHub repository):

```
User: hello
AI system: Hello! How can I assist you today?
User: I'm planning a 1-day trip in Venice. What should I visit?
AI system: That sounds like a wonderful plan! In Venice, there are several must-visit att
1. St. Mark's Square (Piazza San Marco): [...] Enjoy your trip to Venice!
User: thanks! I'm planning to be around also the next day, and I love hiking. Do you have
AI system: Certainly! If you enjoy hiking and want to explore the natural beauty around V
1. The Dolomites: [...]
User: which one is closer to Milan?
AI system: If you're looking for a hiking destination closer to Milan, the best option wo
```

As you can see, now the AI assistant is capable of keeping track of the whole conversation. In the next section, we are going to add yet another layer of complexity: an external knowledge base.

Adding non-parametric knowledge

Imagine that you also want your GlobeBotter to have access to exclusive documentation about itineraries that are not part of its parametric knowledge.

To do so, we can either embed the documentation in a VectorDB or directly use a retriever to do the job. In this case, we will use a vector-store-backed retriever using a particular chain, `ConversationalRetrievalChain`. This type of chain leverages a retriever over the provided knowledge base that has the chat history, which can be passed as a parameter using the desired type of memory previously seen.

With this goal in mind, we will use a sample Italy travel guide PDF downloaded from <https://www.minube.net/guides/italy>.

The following Python code shows how to initialize all the ingredients we need, which are:

- **Document Loader:** Since the document is in PDF format, we will use `PyPDFLoader`.
- **Text splitter:** We will use a `RecursiveCharacterTextSplitter`, which splits text by recursively looking at characters to find one that works.
- **Vector store:** We will use the `FAISS` VectorDB.
- **Memory:** We will use a `ConversationBufferMemory`.
- **LLMs:** We will use the `gpt-3.5-turbo` model for conversations.
- **Embeddings:** We will use the `text-embedding-ada-002`.

Let's take a look at the code:

```

from langchain.llms import OpenAI
from langchain.chat_models import ChatOpenAI
from langchain.embeddings.openai import OpenAIEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS
from langchain.document_loaders import PyPDFLoader
from langchain.chains import ConversationalRetrievalChain
from langchain.memory import ConversationBufferMemory
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1500,
    chunk_overlap=200
)
raw_documents = PyPDFLoader('italy_travel.pdf').load()
documents = text_splitter.split_documents(raw_documents)
db = FAISS.from_documents(documents, OpenAIEmbeddings())
memory = ConversationBufferMemory(
    memory_key='chat_history',
    return_messages=True
)
llm = ChatOpenAI()

```

Let's now interact with the chain:

```

qa_chain = ConversationalRetrievalChain.from_llm(llm, retriever=db.as_retriever(), memory=
qa_chain.run({'question': 'Give me some review about the Pantheon'})

```

The following is the output (I'm reporting a truncated version. You can see the whole output in the book's GitHub repository):

```

> Entering new StuffDocumentsChain chain...
> Entering new LLMChain chain...
Prompt after formatting:
System: Use the following pieces of context to answer the users question.
If you don't know the answer, just say that you don't know, don't try to make up an answer
-----
cafes in the square. The most famous are the Quadri and
Florian.
Piazza San Marco,
Venice
4
Historical Monuments
Pantheon
Miskita:
"Angelic and non-human design," was how
Michelangelo described the Pantheon 14 centuries after its
construction. The highlights are the gigantic dome, the upper
eye, the sheer size of the place, and the harmony of the
whole building. We visited with a Roman guide which is
...
> Finished chain.
'Miskita:\n"Angelic and non-human design," was how Michelangelo described the Pantheon 14

```

Note that, by default, the `ConversationalRetrievalChain` uses a prompt template called `CONDENSE_QUESTION_PROMPT`, which merges the last user's query with the chat history, so that it results as just one query to the retriever. If you want to pass a custom prompt, you can do so using the `condense_question_prompt` parameter in the `ConversationalRetrievalChain.from_llm` module.

Even though the bot was able to provide an answer based on the documentation, we still have a limitation. In fact, with such a configuration, our GlobeBotter will only look at the provided documentation, but what if we want it to also use its parametric knowledge? For example, we might want the bot to be able to understand whether it could integrate with the provided documentation or simply answer *freely*. To do so, we need to make our GlobeBotter *agentic*, meaning that we want to leverage the LLM's reasoning capabilities to orchestrate and invoke the available tools without a fixed order, but rather following the best approach given the user's query.

To do so, we will use two main components:

- `create_retriever_tool`: This method creates a custom tool that acts as a retriever for an agent. It will need a database to retrieve from, a name, and a short description, so that the model can understand when to use it.
- `create_conversational_retrieval_agent`: This method initializes a conversational agent that is configured to work with retrievers and chat models. It will need an LLM, a list of tools (in our case, the retriever), and a memory key to keep track of the previous chat history.

The following code illustrates how to initialize the agent:

```
from langchain.agents.agent_toolkits import create_retriever_tool
tool = create_retriever_tool(
    db.as_retriever(),
    "italy_travel",
    "Searches and returns documents regarding Italy."
)
tools = [tool]
memory = ConversationBufferMemory(
    memory_key='chat_history',
    return_messages=True
)
from langchain.agents.agent_toolkits import create_conversational_retrieval_agent
from langchain.chat_models import ChatOpenAI
llm = ChatOpenAI(temperature = 0)
agent_executor = create_conversational_retrieval_agent(llm, tools, memory_key='chat_history')
```

Great, now let's see the thought process of the agent with two different questions (I will report only the chain of thoughts and truncate the output, but you can find the whole code in the GitHub repo):

```
agent_executor({"input": "Tell me something about Pantheon"})
```

Here is the output:

```
> Entering new AgentExecutor chain...
Invoking: `italy_travel` with `Pantheon`
[Document(page_content='cafes in the square. The most famous are the Quadri and\nFlorian.
> Finished chain.
```

Let's now try with a question not related to the document:

```
output = agent_executor({"input": "what can I visit in India in 3 days?"})
```

The following is the output that we receive:

```
> Entering new AgentExecutor chain...
In India, there are numerous incredible places to visit, each with its own unique attracti
1. Delhi: Start your trip in the capital city of India, Delhi. [...]
> Finished chain.
```

As you can see, when I asked the agent something about Italy, it immediately invoked the provided document, while this was not done in the last question.

The last thing we want to add to our GlobeBotter is the capability to navigate the web, since, as travelers, we want to have up-to-date information about the country we are traveling to. Let's implement it with LangChain's tools.

Adding external tools

The tool we are going to add here is the Google SerpApi tool, so that our bot will be able to navigate the internet.

Note



SerpApi is a real-time API designed to access Google search results. It simplifies the process of data scraping by handling complexities such as managing proxies, solving CAPTCHAs, and parsing structured data from search engine results pages.

LangChain offers a pre-built tool that wraps SerpApi to make it easier to integrate it within your agents. To enable SerpApi, you need to sign in at https://serpapi.com/users/sign_up, then go to the dashboard under the tab **API key**.

Since we don't want our GlobeBotter to be focused only on the web, we will add the SerpApi tool to the previous one, so that the agent will be able to pick the most useful tool to answer the question – or use no tool if not necessary.

Let's initialize our tools and agent (you learned about this and other LangChain components in *Chapter 5*):

```
from langchain import SerpAPIWrapper
import os
from dotenv import load_dotenv
load_dotenv()
os.environ["SERPAPI_API_KEY"]
search = SerpAPIWrapper()
tools = [
    Tool.from_function(
        func=search.run,
        name="Search",
        description="useful for when you need to answer questions about current events"
    ),
    create_retriever_tool(
        db.as_retriever(),
        "italy_travel",
        "Searches and returns documents regarding Italy."
    )
]
```



```
)
]
agent_executor = create_conversational_retrieval_agent(llm, tools, memory_key='chat_history')
```

Great, now let's test it with three different questions (here, again, the output has been truncated):

- “What can I visit in India in 3 days?”

```
> Entering new AgentExecutor chain...
India is a vast and diverse country with numerous attractions to explore. While it may
1. Delhi: Start your trip in the capital city of India, Delhi. [...]
> Finished chain.
```

In this case, the model doesn't need external knowledge to answer the question, hence it is responding without invoking any tool.

- “What is the weather currently in Delhi?”

```
> Entering new AgentExecutor chain...
Invoking: `Search` with `{'query': 'current weather in Delhi'}`
Current Weather · 95°F Mostly sunny · RealFeel® 105°. Very Hot. RealFeel Guide. Very Hot
> Finished chain.
```

Note how the agent is invoking the search tool; this is due to the reasoning capability of the underlying gpt-3.5-turbo model, which captures the user's intent and dynamically understands which tool to use to accomplish the request.

- “I'm traveling to Italy. Can you give me some suggestions for the main attractions to visit?”

```
> Entering new AgentExecutor chain...
Invoking: `italy_travel` with `{'query': 'main attractions in Italy'}`
[Document(page_content='ITALY\nMINUBE TRAVEL GUIDE\nThe best must-see places for your trip
Here are some suggestions for main attractions in Italy:
1. Parco Sempione, Milan: This is one of the most important parks in Milan. It offers a
> Finished chain.
```

Note how the agent is invoking the document retriever to provide the preceding output.

Overall, our GlobeBotter is now able to provide up-to-date information, as well as retrieving specific knowledge from curated documentation. The next step will be that of building a front-end. We will do so by building a web app using Streamlit.

Developing the front-end with Streamlit

Streamlit is a Python library that allows you to create and share web apps. It is designed to be easy and fast to use, without requiring any front-end experience or knowledge. You can write your app in pure Python, using simple commands to add widgets, charts, tables, and other elements.

In addition to its native capabilities, in July 2023, Streamlit announced an initial integration and its future plans with LangChain. At the core of this initial integration, there is the ambition of making it easier to build a GUI for conversational applications, as well as showing all the steps LangChain's agents take before producing the final response.

To achieve this goal, the main module that Streamlit introduced is the Streamlit callback handler. This module provides a class called `StreamlitCallbackHandler` that implements the `BaseCallbackHandler` interface from LangChain. This class can handle various events that occur during the execution of a LangChain pipeline, such as tool start, tool end, tool error, LLM token, agent action, agent finish, etc.

The class can also create and update Streamlit elements, such as containers, expanders, text, progress bars, etc., to display the output of the pipeline in a user-friendly way. You can use the Streamlit callback handler to create Streamlit apps that showcase the capabilities of LangChain and interact with the user through natural language. For example, you can create an app that takes a user prompt and runs it through an agent that uses different tools and models to generate a response. You can use the Streamlit callback handler to show the agent's thought process and the results of each tool in real time.

To start building your application, you need to create a `.py` file to run in your terminal via `streamlit run file.py`. In our case, the file will be named `globebotter.py`.

The following are the main building blocks of the application:

1. Setting the configuration of the webpage:

```
import streamlit as st
st.set_page_config(page_title="GlobeBotter", page_icon="🌐")
st.header('🌐 Welcome to Globebotter, your travel assistant with Internet access. What ;
```

2. Initializing the LangChain backbone components we need. The code is the same as the one in the previous section, so I will share here only the initialization code, without all the preliminary steps:

```
search = SerpAPIWrapper()
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1500,
    chunk_overlap=200
)
raw_documents = PyPDFLoader('italy_travel.pdf').load()
documents = text_splitter.split_documents(raw_documents)
db = FAISS.from_documents(documents, OpenAIEmbeddings())
memory = ConversationBufferMemory(
    return_messages=True,
    memory_key="chat_history",
    output_key="output"
)
llm = ChatOpenAI()
tools = [
    Tool.from_function(
        func=search.run,
        name="Search",
        description="useful for when you need to answer questions about current events"
```

```

    ),
    create_retriever_tool(
        db.as_retriever(),
        "italy_travel",
        "Searches and returns documents regarding Italy."
    )
]
agent = create_conversational_retrieval_agent(llm, tools, memory_key='chat_history', ve

```

3. Setting the input box for the user with a placeholder question:

```

user_query = st.text_input(
    "***Where are you planning your next vacation?***",
    placeholder="Ask me anything!"
)

```

4. Setting Streamlit's session states. Session state is a way to share variables between reruns, for each user session. In addition to the ability to store and persist state, Streamlit also exposes the ability to manipulate state using callbacks. Session state also persists across apps inside a multipage app. You can use the session state API to initialize, read, update, and delete variables in the session state. In the case of our GlobeBotter, we want two main states: `messages` and `memory`:

```

if "messages" not in st.session_state:
    st.session_state["messages"] = [{"role": "assistant", "content": "How can I help yo
if "memory" not in st.session_state:
    st.session_state['memory'] = memory

```

5. Making sure to display the whole conversation. To do so, I created a for loop that iterates over the list of messages stored in `st.session_state["messages"]`. For each message, it creates a Streamlit element called `st.chat_message` that displays a chat message in a nice format:

```

for msg in st.session_state["messages"]:
    st.chat_message(msg["role"]).write(msg["content"])

```

6. Configuring the AI assistant to respond when given a user's query. In this first example, we will keep the whole chain visible and printed to the screen:

```

if user_query:
    st.session_state.messages.append({"role": "user", "content": user_query})
    st.chat_message("user").write(user_query)
    with st.chat_message("assistant"):
        st_cb = StreamlitCallbackHandler(st.container())
        response = agent(user_query, callbacks=[st_cb])
        st.session_state.messages.append({"role": "assistant", "content": response})
        st.write(response)

```

7. Finally, adding a button to clear the history of the conversation and start from scratch:

```

if st.sidebar.button("Reset chat history"):
    st.session_state.messages = []


```


The final product looks as follows:


Welcome to Globebotter, your travel assistant with Internet access. What are you planning for your next trip?

Where are you planning your next vacation?

What is the temperature today in Rome?

 How can I help you?

 What is the temperature today in Rome?

 ☒ Search: temperature today in Rome

```
{
  "input": "what is the temperature today in Rome?"
  "chat_history": [...]
  "output": "The current temperature in Rome is 97 °F. It is sunny."
  "intermediate_steps": [...]
}
```

Figure 6.2: Front-end of GlobeBotter with Streamlit

From the expander, we can see that the agent used the `Search` tool (provided with the `SerpApi`). We can also expand `chat_history` or `intermediate_steps` as follows:

 ☒ Search: temperature today in Rome

```
{
  "input": "what is the temperature today in Rome?"
  "chat_history": [...]
  "output": "The current temperature in Rome is 97 °F. It is sunny."
  "intermediate_steps": [...]
    0 : [
      0 :
        "_FunctionsAgentAction(tool='Search', tool_input='temperature today in Rome', log='\nInvoking: `Search` with `temperature today in Rome`\n\n', message_log=[AIMessage(content='', additional_kwargs={'function_call': {'name': 'Search', 'arguments': {'\n__arg1': 'temperature today in Rome'\n}}}, example=False)])"
      1 :
        "TodayHourly14 DaysPastClimate. Currently: 97 °F. Sunny. (Weather station: Rome Urbe Airport, Italy). See more current weather."
    ]
}
```

Figure 6.3: Example of Streamlit expander

Of course, we can also decide to only show the output rather than the whole chain of thoughts, by specifying in the code to return only `response['output']`. You can see the whole code in the book's GitHub repository.

Before we wrap up, let's discuss how you can give your users a streaming experience while interacting with your chatbot. You can leverage the `BaseCallbackHandler` class to create a custom callback handler in your Streamlit app:

```

from langchain.callbacks.base import BaseCallbackHandler
from langchain.schema import ChatMessage
from langchain_openai import ChatOpenAI
import streamlit as st

class StreamHandler(BaseCallbackHandler):
    def __init__(self, container, initial_text=""):
        self.container = container
        self.text = initial_text

    def on_llm_new_token(self, token: str, **kwargs) -> None:
        self.text += token
        self.container.markdown(self.text)

```

The `StreamHandler` is designed to capture and display streaming data, such as text or other content, in a designated container. Then, you can use it as follows in your Streamlit app, making sure to set `streaming=True` while initializing your OpenAI LLM.

```

with st.chat_message("assistant"):
    stream_handler = StreamHandler(st.empty())
    llm = ChatOpenAI(streaming=True, callbacks=[stream_handler])
    response = llm.invoke(st.session_state.messages)
    st.session_state.messages.append(ChatMessage(role="assistant", content=response.co

```

You can refer to the original code on LangChain's GitHub repo at https://github.com/langchain-ai/streamlit-agent/blob/main/streamlit_agent/basic_streaming.py.

Summary

In this chapter, we approached the end-to-end implementation of a conversational application, leveraging LangChain's modules and progressively adding layers of complexity. We started with a plain vanilla chatbot with no memory, then moved on to more complex systems with the ability to keep traces of past interactions. We've also seen how to add non-parametric knowledge to our application with external tools, making it more "agentic" so that it is able to determine which tool to use, depending on the user's query. Finally, we introduced Streamlit as the front-end framework to build the web app for our GlobeBotter.

In the next chapter, we will focus on a more specific domain where LLMs add value and demonstrate emerging behaviors, that is, recommendation systems.

References

- Example of a context-aware chatbot. https://github.com/shashankdeshpande/langchain-chatbot/blob/master/pages/2_%E2%AD%A0_context_aware_chatbot.py
- Knowledge base for the AI travel assistant. <https://www.minube.net/guides/italy>
- LangChain repository. <https://github.com/langchain-ai>

Join our community on Discord

Join our community's Discord space for discussions with the author and other readers:

<https://packt.link/llm>

