Chapter 7. Advanced Text Generation Techniques and Tools

Introduction

Prompt engineering can highly improve the accuracy of the generated text. In addition to prompt engineering, What can we do to further enhance the experience and output that we get from the LLM without needing to fine-tune the model itself?

In this chapter, we will explore several such methods and concepts for improving the quality of the generated text:

- Model I/O - Loading and working with LLMs
- Memory - Helping LLMs to remember
- Agents - Combining complex behavior with external tools
- Chains - Connecting methods and modules

These methods are all integrated with the LangChain framework that will help us easily use these advanced techniques throughout this chapter.

LangChain

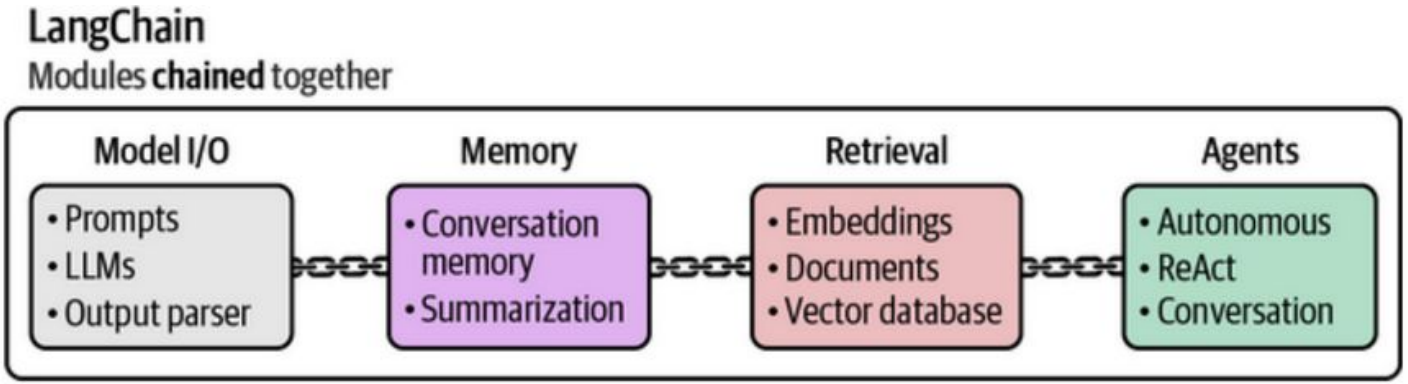

Figure 7-1. LangChain is a complete framework for using LLMs. It has modular components that can be chained together to allow for complex LLM systems.

Model I/O: Loading Quantized Models with LangChain

- Bits, a series of 0s and 1s, represent values by encoding them in binary form. More bits result in a wider range of values but requires more memory to store those values.
- Quantization reduces the number of bits required to represent the parameters of an LLM while attempting to maintain most of the original information. This comes with some loss in precision but often makes up for it as the model is much faster to run, requires less VRAM, and is often almost as accurate as the original.

Quantization is a similar process that reduces the precision of a value (e.g., removing seconds) without removing vital information (e.g., retaining hours and minutes).

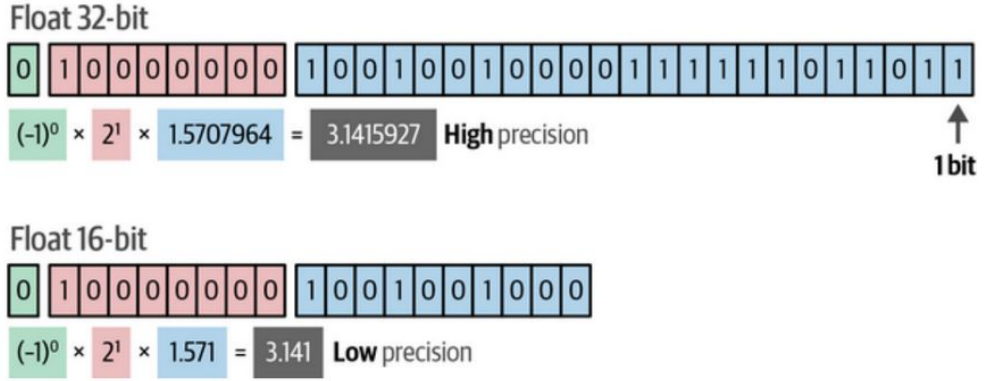

Figure 7-2. Attempting to represent pi with float 32-bit and float 16-bit representations. Notice the lowered accuracy when we halve the number of bits.

Chains: Extending the Capabilities of LLMs

Some applications are more involved and require lengthy or complex prompts to generate a response that captures those intricate details.

we could break this complex prompt into smaller subtasks that can be run sequentially. This would require multiple calls to the LLM but with smaller prompts and intermediate outputs as shown in Figure 7-7.

Instead of using a single chain, we link chains where each link deals with a specific subtask.

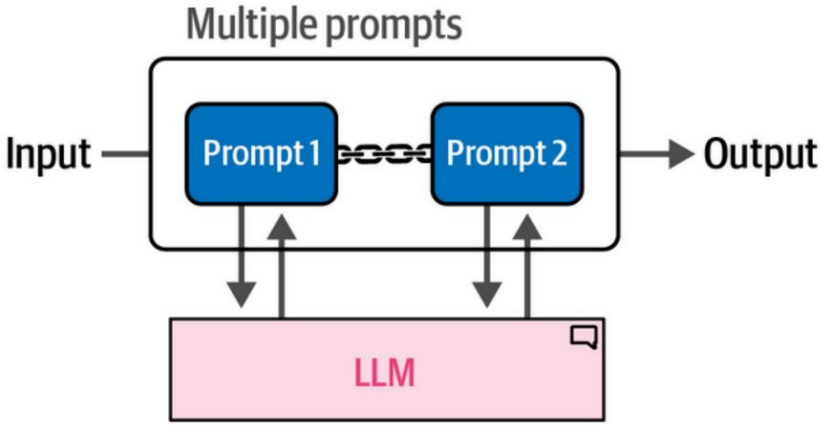

Figure 7-7. With sequential chains, the output of a prompt is used as the input for the next prompt.

Chains with a Single Prompt

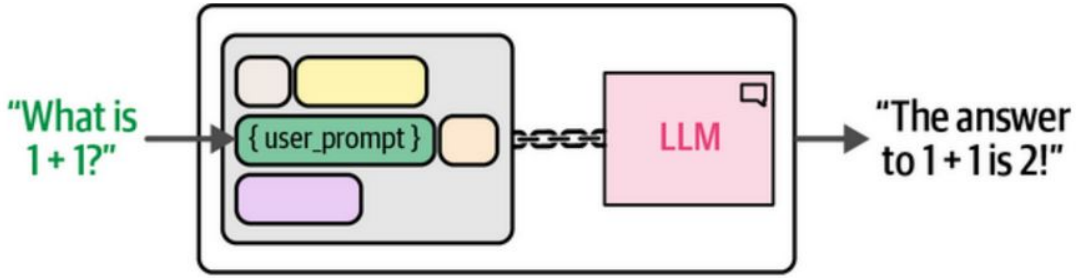

Figure 7-6. An example of a single chain using Phi-3's template.

Chains with multiple prompts.

Instead of generating everything in one go, we create a chain that only requires a single input by the user and then sequentially generates the three components. This process is illustrated in Figure 7-8.

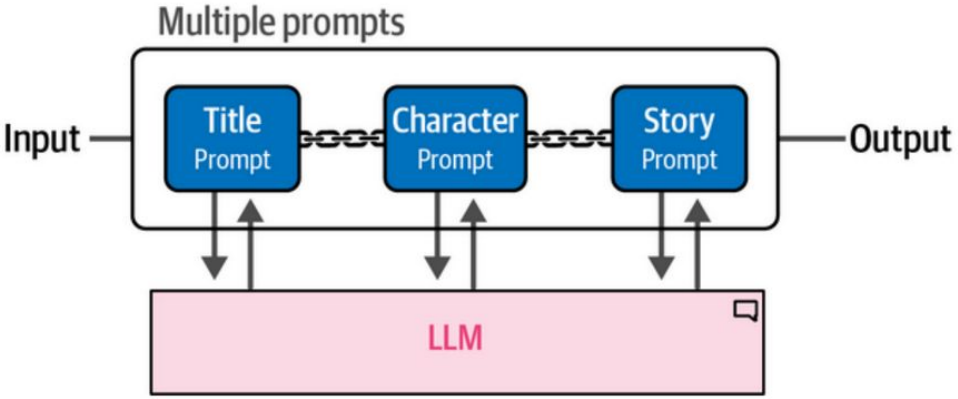

Figure 7-8. The output of the title prompt is used as the input of the character prompt. To generate the story, the output of all previous prompts is used.

Memory: Helping LLMs to Remember Conversations

When we are using LLMs out of the box, they will not remember what was being said in a conversation.
You can share your name in one prompt but it will have forgotten it by the next prompt.
Let's illustrate this phenomenon with an example using the basic_chain we created before. First, we tell the LLM our name:

```
# Let's give the LLM our name
basic_chain.invoke({"input_prompt": "Hi! My name is Maarten. What
is 1 + 1?"})
```

```
Hello Maarten! The answer to 1 + 1 is 2.
```

Next, we ask it to reproduce the name we have given it:

```
basic_chain.invoke({"input_prompt": "What is my name?"})
```

```
I'm sorry, but as a language model, I don't have the ability to
know personal information about individuals. You can provide
the name you'd like to know more about, and I can help you with
information or general inquiries related to it.
```

Unfortunately, the LLM does not know the name we gave it. The reason for this forgetful behavior is that these models are stateless—they have no memory of any previous conversation!

To make these models stateful, we can add specific types of memory to the chain that we created earlier. In this section, we will go through two common methods for helping LLMs to remember conservations:

- Conversation buffer
- Conversation summary

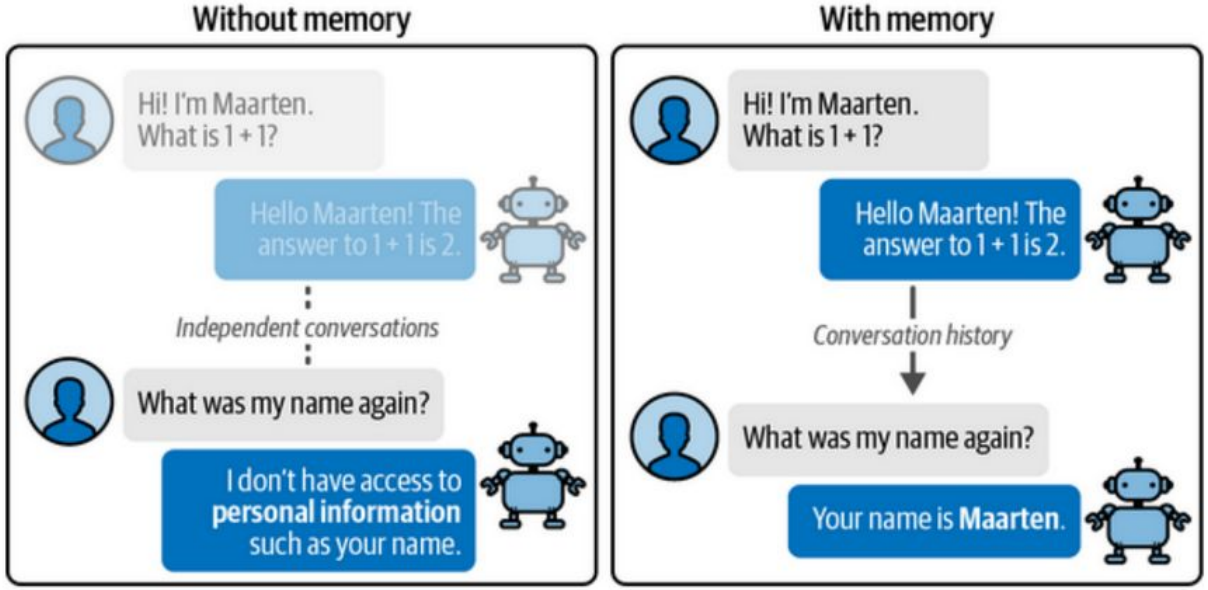

Figure 7-9. An example of a conversation between an LLM with memory and without memory.

Conversation Buffer

One of the most intuitive forms of giving LLMs memory is simply reminding them exactly what has happened in the past. As illustrated in Figure 7-10, we can achieve this by copying the full conversation history and pasting that into our prompt.

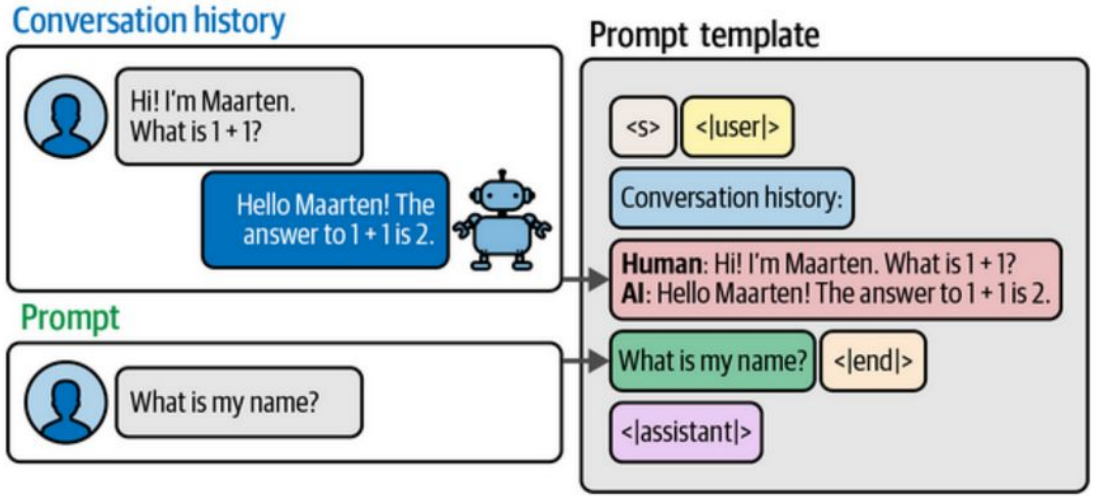

Figure 7-10. We can remind an LLM of what previously happened by simply appending the entire conversation history to the input prompt.

Notice that we added an additional input variable, namely chat_history. This is where the conversation history will be given before we ask the LLM our question.

Implementing Conversionmemory in LangChain

```
# Create an updated prompt template to include a chat history
template = """<s><|user|>Current conversation:{chat_history}

{input_prompt}<|end|>
<|assistant|>"""

prompt = PromptTemplate(
    template=template,
    input_variables=["input_prompt", "chat_history"]
)
```

```
from langchain.memory import ConversationBufferMemory

# Define the type of memory we will use
memory = ConversationBufferMemory(memory_key="chat_history")

# Chain the LLM, prompt, and memory together
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm,
    memory=memory
)
```

Windowed Conversion Buffer

In our previous example, we essentially created a chatbot. You could talk to it and it remembers the conversation you had thus far. However, as the size of the conversation grows, so does the size of the input prompt until it exceeds the token limit.

One method of minimizing the context window is to use the last k conversations instead of maintaining the full chat history. In LangChain, we can use ConversationBufferWindowMemory to decide how many conversations are passed to the input prompt:

```
from langchain.memory import ConversationBufferWindowMemory

# Retain only the last 2 conversations in memory
memory = ConversationBufferWindowMemory(k=2,
memory_key="chat_history")

# Chain the LLM, prompt, and memory together
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm,
    memory=memory
)
```

Conversation Summary

Although a solution would be to use an LLM with a larger context window, these tokens still need to be processed before generation tokens, which can increase compute time. Instead, let's look toward a more sophisticated technique, ConversationSummaryMemory. As the name implies, this technique summarizes an entire conversation history to distill it into the main points.

This summarization process is enabled by another LLM that is given the conversation history as input and asked to create a concise summary. A nice advantage of using an external LLM is that we are not confined to using the same LLM during conversation. The summarization process is illustrated in Figure 7-12.

This means that whenever we ask the LLM a question, there are two calls:

1. The user prompt
2. The summarization prompt

To use this in LangChain, we first need to prepare a summarization template that we will use as the summarization prompt:

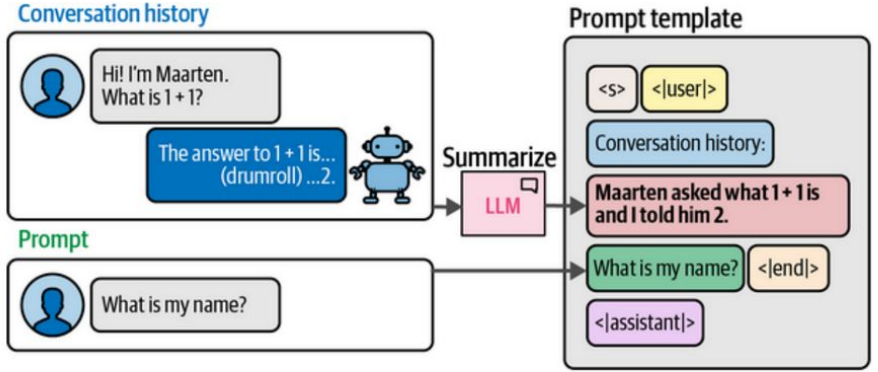

Figure 7-12. Instead of passing the conversation history directly to the prompt, we use another LLM to summarize it first.

This more complex chain including conversation summary is illustrated in Figure 7-13 to give an overview of this additional functionality.

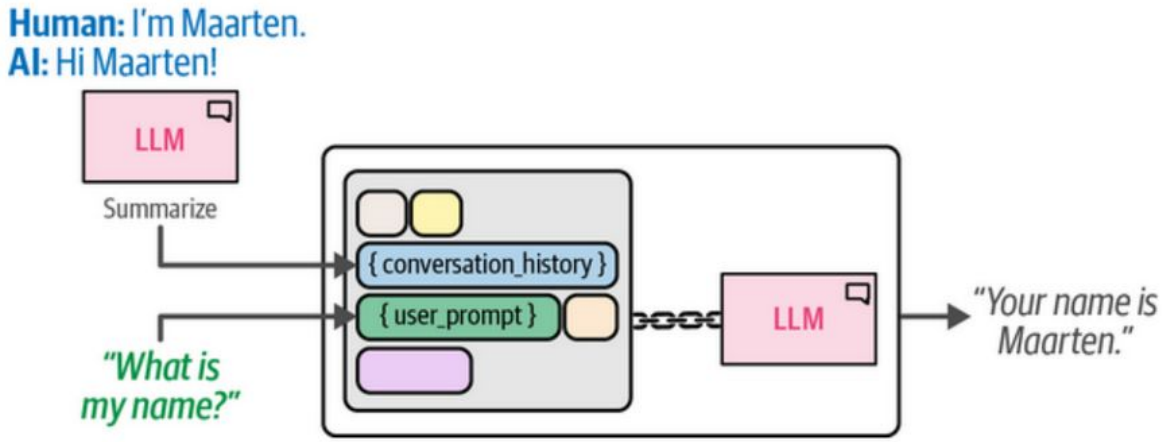

Figure 7-13. We extend the LLM chain with memory by summarizing the entire conversation history before giving it to the input prompt.

Conversation Buffer vs. Conversation Summary

Advantages of Conversation Summary

This summarization helps keep the chat history relatively small without using too many tokens during inference.

Disadvantages of Conversation Summary

since the original question was not explicitly saved in the chat history, the model needed to infer it based on the context. This is a disadvantage if specific information needs to be stored in the chat history. Moreover, multiple calls to the same LLM are needed, one for the prompt and one for the summarization. This can slow down computing time.

Use Cases

Memory Type	Pros	Cons
Conversation Buffer	Easiest implementation; Ensures no information loss within context window	Slower generation speed as more tokens are needed; Only suitable for large-context LLMs; Larger chat histories make information retrieval difficult
Windowed Conversation Buffer	No information loss over the last k interactions; Only captures the last k interactions; No compression of the last k interactions	Large-context LLMs are not needed unless chat history is large
Conversation Summary	Captures the full history; Enables long conversations; Reduces tokens needed to capture full history	An additional call is necessary for each interaction; Quality is reliant on the LLM's summarization capabilities

Agents: Creating a System of LLMs

Thus far, we have created systems that follow a user-defined set of steps to take. One of the most promising concepts in LLMs is their ability to determine the actions they can take. This idea is often called agents, systems that leverage a language model to determine which actions they should take and in what order.

Agents can make use of everything we have seen thus far, such as model I/O, chains, and memory, and extend it further with two vital components:

- Tools that the agent can use to do things it could not do itself
- The agent type, which plans the actions to take or tools to use

Unlike the chains we have seen thus far, agents are able to show more advanced behavior like creating and self-correcting a roadmap to achieve a goal. They can interact with the real world through the use of tools. As a result, these agents can perform a variety of tasks that go beyond what an LLM is capable of in isolation.

Agents

As illustrated in Figure 7-14, the underlying idea of agents is that they utilize LLMs not only to understand our query but also to decide which tool to use and when.

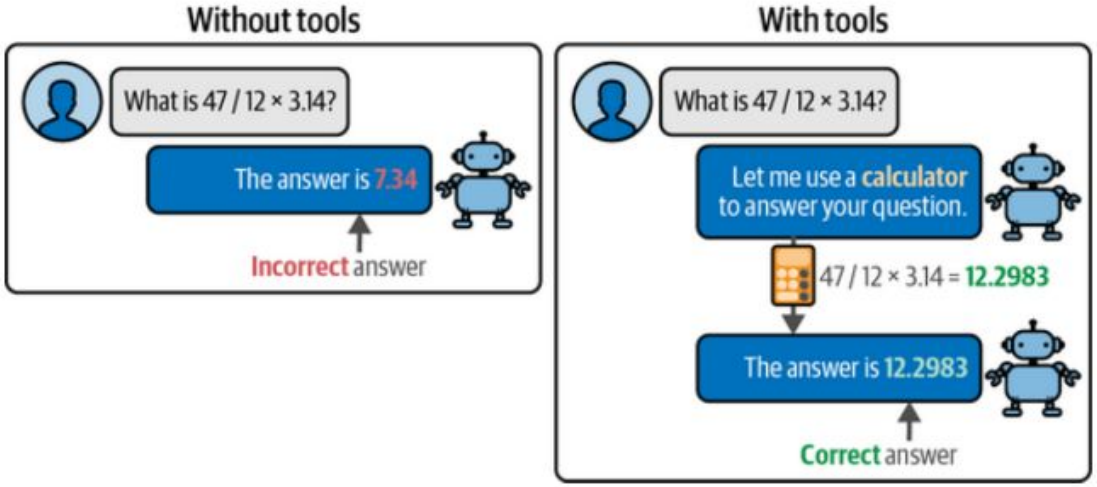

Figure 7-14. Giving LLMs the ability to choose which tools they use for a particular problem results in more complex and accurate behavior

In this example, we would expect the LLM to use the calculator when it faces a mathematical task. Now imagine we extend this with dozens of other tools, like a search engine or a weather API. Suddenly, the capabilities of LLMs increase significantly.

Creating Agents using ReAct in LangChain

ReAct is a powerful framework that combines two important concepts in behavior: reasoning and acting.

ReAct merges these two concepts and allows reasoning to affect acting and actions to affect reasoning. In practice, the framework consists of iteratively following these three steps:

- Thought
- Action
- Observation

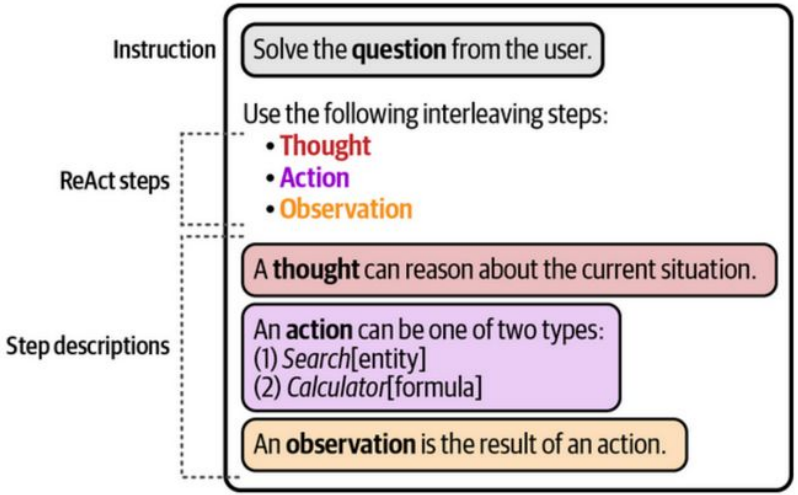

Figure 7-15. An example of a ReAct prompt template.

During this process, the agent describes its thoughts (what it should do), its actions (what it will do), and its observations (the results of the action). It is a cycle of thoughts, actions, and observations that results in the agent's output.

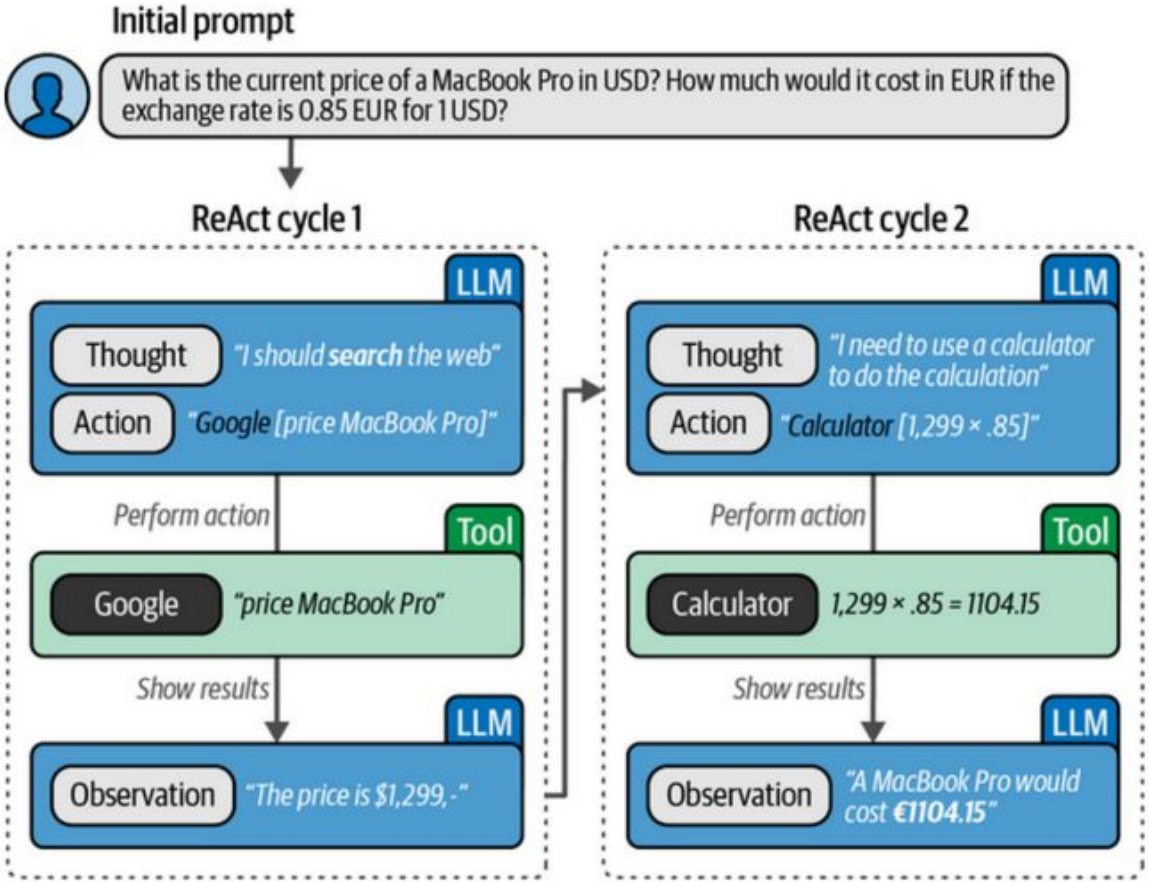

Figure 7-16. An example of two cycles in a ReAct pipeline.