Chapter 7 - Advanced Text Generation Techniques and Tools

Important Topics

1. **Model I/O**: Loading and working with LLMs
2. **Memory**: Helping LLMs to remember
3. **Agents**: Combining complex behavior with external tools
4. **Chains**: Connecting methods and modules

These methods are all integrated with the LangChain framework.

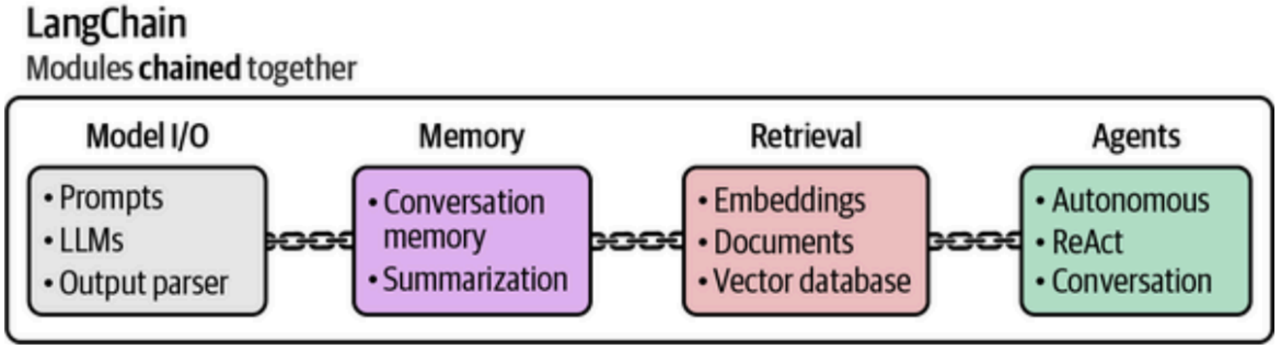


Figure 7-1. LangChain is a complete framework for using LLMs. It has modular components that can be chained together to allow for complex LLM systems.

Model I/O: Loading Quantized Models with LangChain

As in previous chapters, we will be using Phi-3 GGUF (GPT-Generated Unified Format) model variant.

A **GGUF** model represents a compressed version of its original counterpart through a method called **quantization**, which *reduces the number of bits needed to represent the parameters of* *an LLM.*

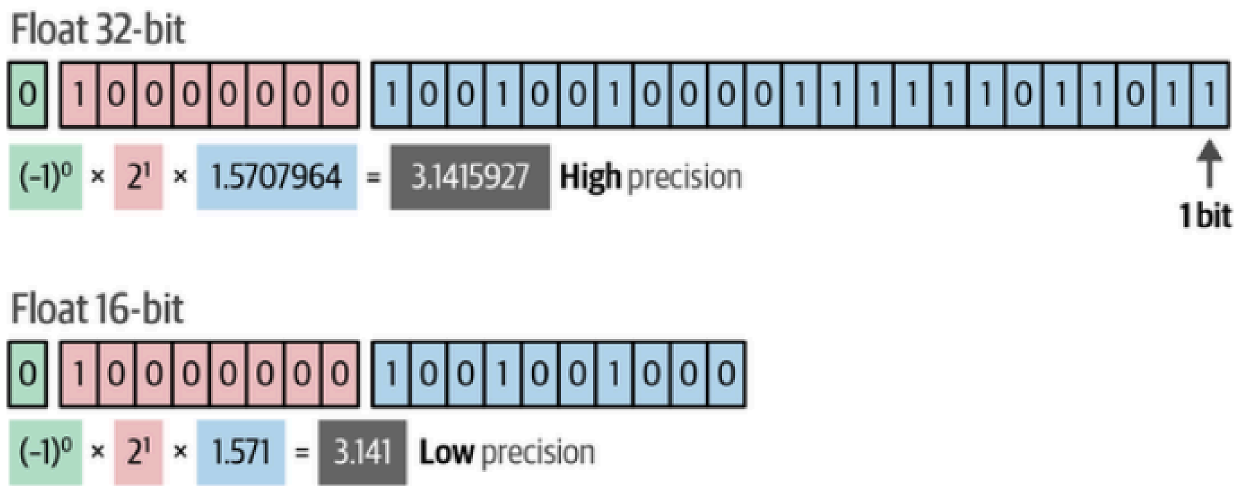


Figure 7-2. Attempting to represent pi with float 32-bit and float 16-bit representations. Notice the

Quantization

Quantization reduces the number of bits required to represent the parameters the parameters of an LLM while attempting most of the original information. This could cause some reduction in precision. But this loss is made up for in speed, and lower resource needs (requires less VRAM).

Know that for the rest of the chapter, we will be using an 8-bit variant of Phi-3 compared to the original 16-bit variant, cutting memory requirements almost in half.

First, we need to download the model. `FP16`, the chosen model, represents the **16-bit** variant.

```
!wget https://huggingface.co.microsoft/Phi-3-mini-4k-instruct-
gguf/resolve/main/Phi-3-mini-4k-instruct-fp16.gguf
```

We use `llama-cpp-python` together with LangChain to load the GGUF file:

```
from langchain import LLamaCpp

# Make sure the model path is correct for your system
llm = LlamaCpp(
    model_path = "Phi-3-mini-4k-instruct-fp16.gguf",
    n_gpu_layers = -1 # -1 means you want all the layers to use the GPU.
    max_tokens = 500,
    n_ctx = 2048, # this defines the context length, discussed in previous
chapters
    seed = 42,
    verbose = False
    )
```

In LangChain, we use the `invoke` function to generate output:

```
llm.invoke("Hi! My name is Maarten. What is 1 + 1?")
```

This outputs:

```
''
```

As you can see, there is no output. Looking back to the previous chapters, we know `Phi-3` requires a specific prompt template. But instead of having to use this template every time, we could instead use **LangChain**'s core functionality, called '*chains*'. Click here for the documentation.

TIP

Although we have been using `Phi-3`, you can choose any LLM. To see the best current models, visit this page.

Chains: Extending the Capabilities of LLMs

Although LLMs can be run in isolation, chains allow for better integration with tools, agent-like behavior, and combination with other tools.

The most basic form of a chain in LangChain is a single chain. Although a chain can vary in complexity and form, it generally connects an LLM with some additional tool, prompt or feature.

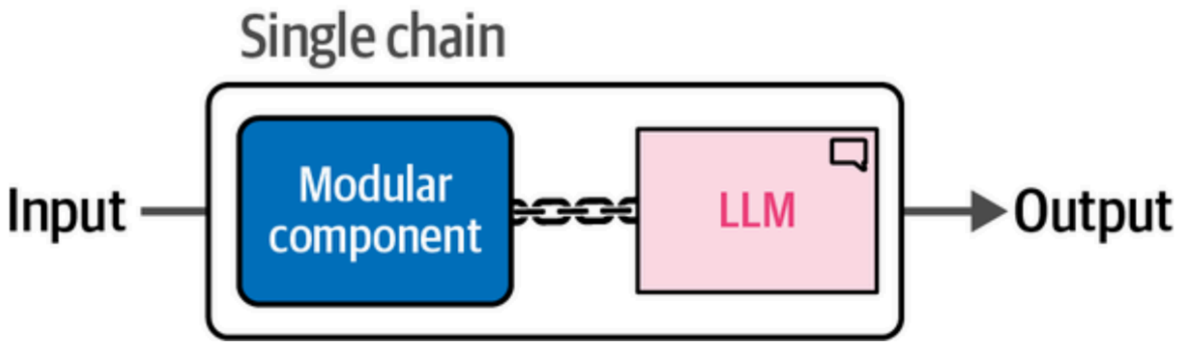


Figure 7-3. A single chain connects some modular component, like a prompt template or external memory, to the LLM.

A Single Link in the Chain: Prompt Template

We start with creating our first chain, namely the prompt template that `Phi-3` expects.

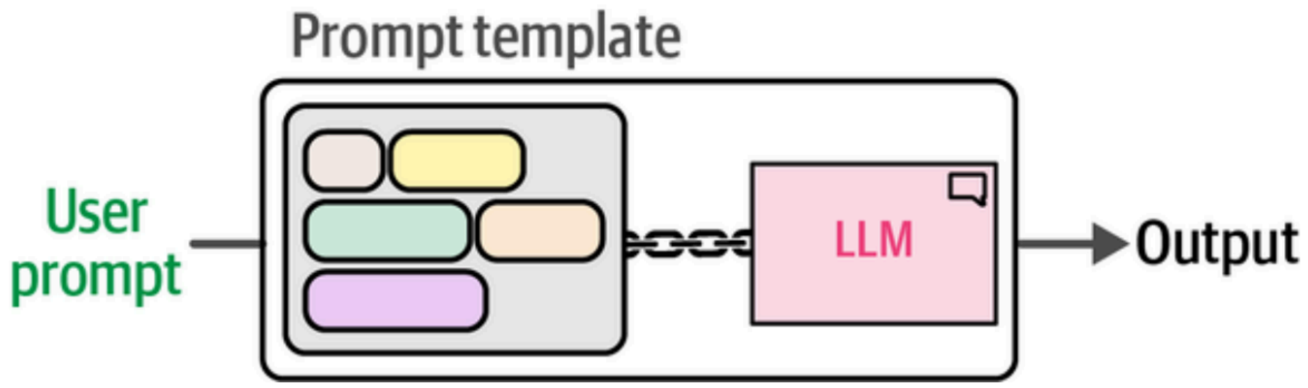


Figure 7-4. By chaining a prompt template with an LLM, we only need to define the input prompts. The template will be constructed for you.

The template for `Phi-3` is comprised of **four main components:**

- `<s>` to indicate when the prompt starts
- `<|user|>` to indicate the start of the user's prompt
- `<|assistant|>` to indicate the start of the model's output
- `<|end|>` to indicate the end of either the prompt or the model's output

The diagram below further illustrates this with an example:

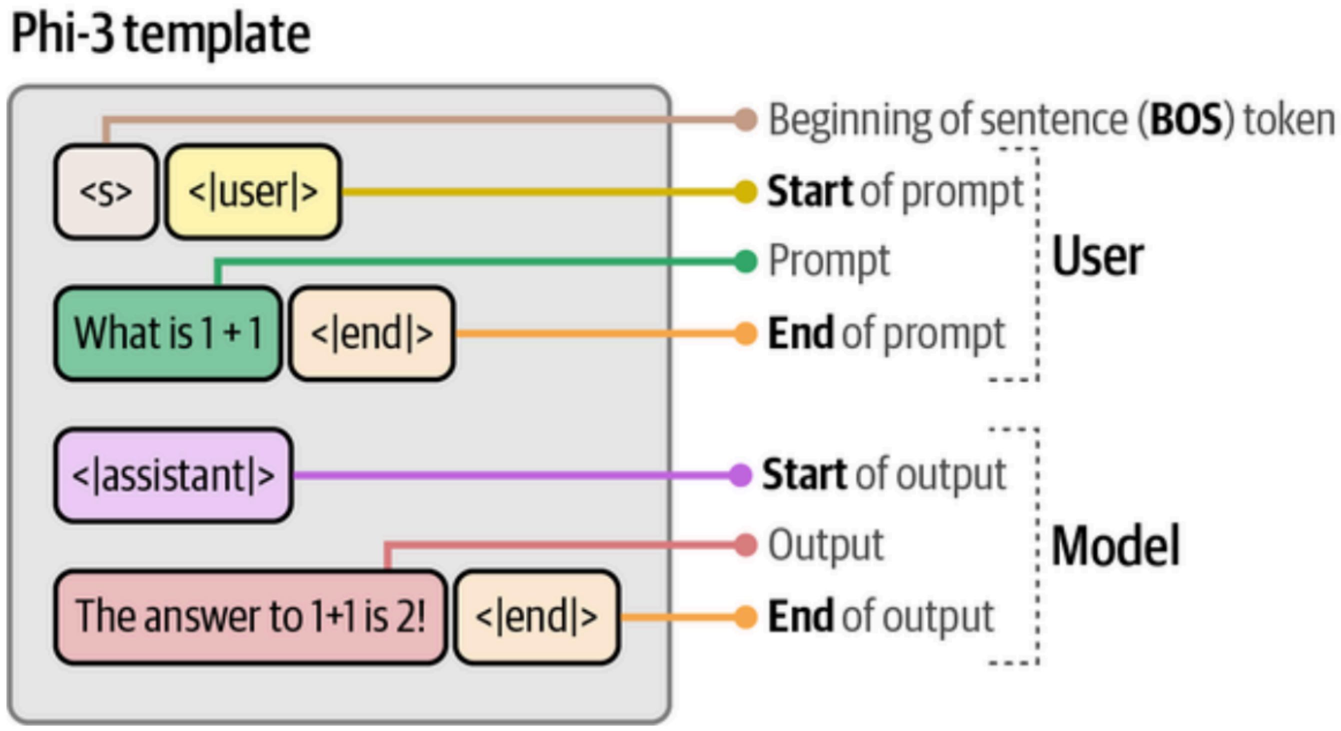


Figure 7-5. The prompt template Phi-3 expects.

Let's try this with a real example:

To generate our simple chain, we first need to create a prompt template that adheres to `Phi-3`'s expected template. Using this template, the model takes in a `system_prompt`, which generally describes what we expect from the LLM.

Then we can use the `input_prompt` to ask the LLM specific questions:

```
from langchain import PromptTemplate

# Create a prompt template with the "input_prompt" variable
template = """<s><|user|>
    {input_prompt}<|end|>
    <|assistant|>"""
prompt = PromptTemplate(
    template = template,
    input_variables = ["input_prompt"]
)
```

To create our first chain, we can use both the prompt that we created and the LLM and chain them together:

```
basic_chain = prompt | llm # llm is the Phi-3 GGUF we defined early in the
chapter
```

To use the chain, we need to use the `invoke` function

```
basic_chain.invoke(
    {
        "input_prompt": "Hi! My name is Maarten. What is 1 + 1?",
    }
)
```

This outputs:

```
The answer to 1 + 1 is 2. It's a basic arithmetic operation where you add
one unit to another, resulting in two units altogether.
```

This output gives us the response without any unnecessary tokens. Now that we have created this chain, we do not have to create the prompt template from scratch every time we use the LLM.

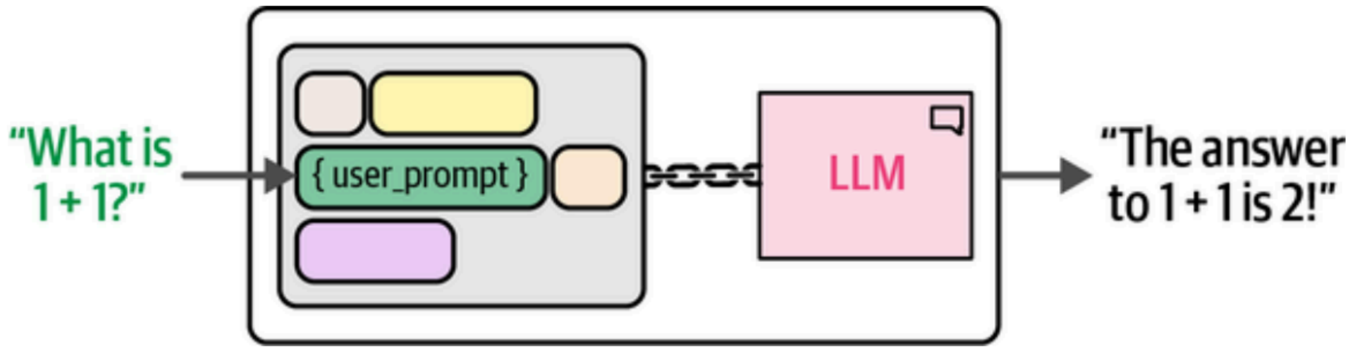


Figure 7-6. An example of a single chain using Phi-3's template.

NOTE

While `Phi-3` and other models expect a specific template, other models like ChatGPT handles the underlying template.

A Chain with Multiple Prompts

In our previous example, we created a single chain consisting of a prompt template an an LLM. Because the previous prompt was pretty straightforward and contains one task (e.g., adding 1 + 1), the LLM was able to handle it with no issues.

This is not always the case. Some prompts could be more complex and contain multiple tasks and dimensions. Giving a this as an input to an LLM could produce poor results.

Prompt chaining, as seen in Chapter 6, addresses this.

Instead of providing an LLM with a long, complex prompt, we break it down to multiple chained prompts. If you remember, the example in chapter 6 was asking the LLM to generate a *name*, a *slogan*, and a *business plan*.

Below, this method is illustrated:

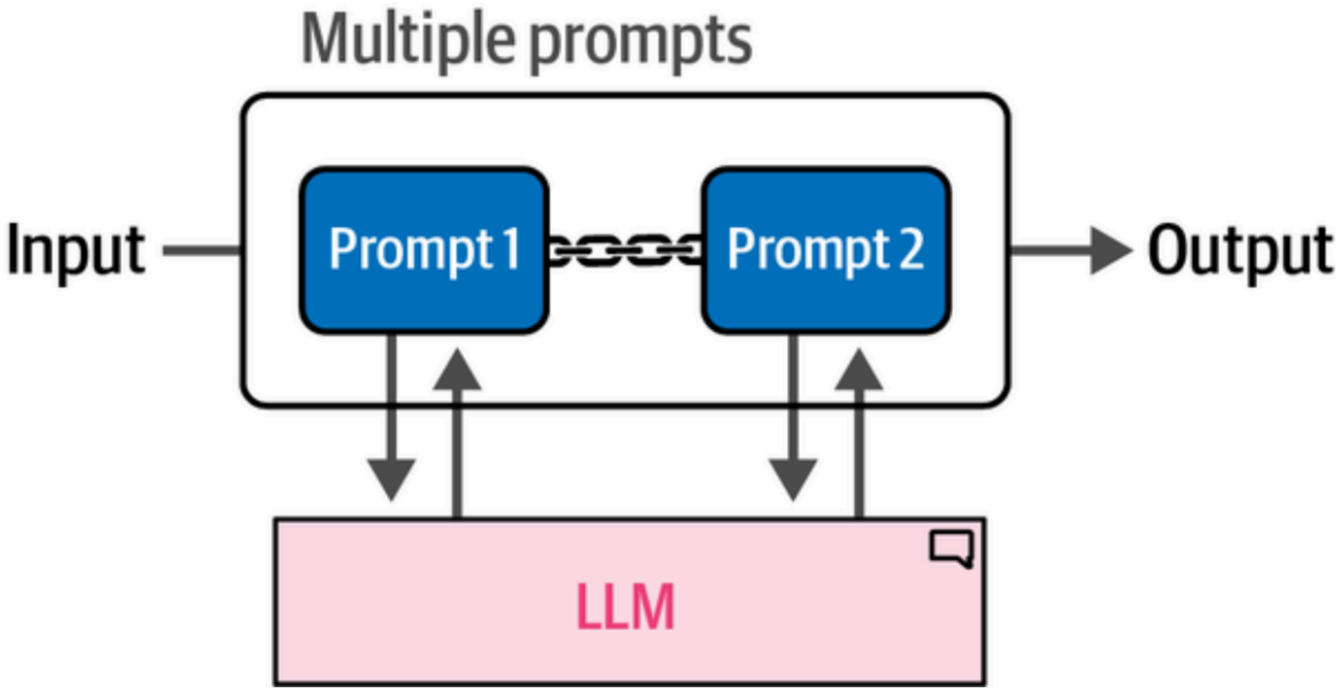


Figure 7-7. With sequential chains, the output of a prompt is used as the input for the next prompt.

Let's try a new example. Assume we want to generate a story that has three components:

- A title
- A description
- A summary of the story

To generate that story, we use LangChain to describe the first component, namely the title. The first link is the only component that requires user input, we define the "summary".

```
from langchain import LLMChain

# Create a chain for the title of our story
template = """<s><|user|>
Create a title for a story about {summary}. Only return the title.
<|end|>
<|assistant|>"""

title_prompt = PromptTemplate(template=template, input_variables=
["summary"])
title = LLMChain(llm=llm, prompt = title_pompt, output_key = title)
```

Let’s run an example:

```
title.invoke({"summary": "a girl that lost her mother"})
```

This outputs:

```
{'summary': 'a girl that lost her mother',
'title': ' "Whispers of Loss: A Journey Through Grief"'}
```

Using the generated output, let's generate the next component, the "description".

```
# Create a chain for the character description using the summary and title

template = """<s><|user|>
Describe the main character of a story about {summary} with the
title {title}. Use only two sentences.<|end|>
<|assistant|>"""

character_prompt = PromptTemplate(
    template=template, input_variables=["summary", "title"]
)
character = LLMChain(llm=llm, prompt=character_prompt,
output_key="character")
```

Now let's create a short description of the story using all the components we have.

```
# Create a chain for the story using the summary, title, and character
description
template = """<s><|user|>
Create a story about {summary} with the title {title}. The main
character is: {character}. Only return the story and it cannot be
longer than one paragraph. <|end|>
<|assistant|>"""

story_prompt = PromptTemplate(
    template=template, input_variables=["summary", "title",
    "character"]
)
story = LLMChain(llm=llm, prompt=story_prompt, output_key="story")
```

Now that we have created all three components, we can link them together to create our full chain:

```
# Combine all three components to generate a story
llm_chain = title | character | story
```

This outputs:

```
{'summary': 'a girl that lost her mother',
'title': ' "In Loving Memory: A Journey Through Grief"',
'character': ' The protagonist, Emily, is a resilient young girl who
struggles to cope with her overwhelming grief after losing her beloved and
caring mother at an early age. As she embarks on a journey of self-discovery
and healing, she learnsvaluable life lessons from the memories and wisdom
shared by those around her.',
'story': " In Loving Memory: A Journey Through Grief revolves around Emily,
a resilient young girl who loses her beloved mother at an early age.
Struggling to cope with overwhelming grief, she embarks on a journey of
self-discovery and healing, drawing strength from the cherished memories and
wisdom shared by those around her. Through this transformative process,
Emily learns valuable life lessons about resilience, love, and the power of
human connection, ultimately finding solace in honoring her mother's legacy
while embracing a newfound sense of inner peace amidst the painful loss."}
```

Memory: Helping LLMs to Remember Conversations

When we use LLMs out of the box (basic inference), they will not remember what was being said in a conversation. Any information shared in one prompt will not be remembered in the next.

Let's test this with code:

```
# 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.
```

Now let's see if it can remember the name we have provided.

```
# Next, we ask the LLM to reproduce the name
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.
```

The reason for this behavior is that these models are stateless. It does not store any memory of past interactions. **Every transaction is handled as if it were the first time.**

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 conversations:

- **Conversation buffer**
- **Conversation summary**

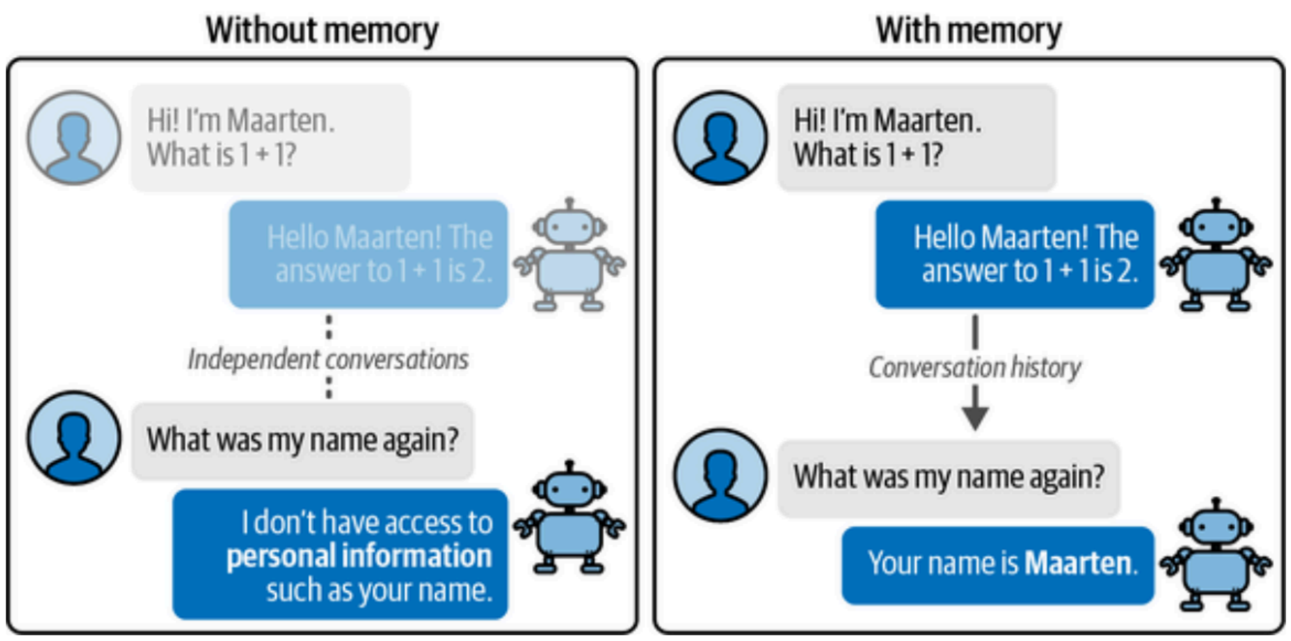

Without memory
Hi! I'm Maarten. What is 1 + 1?
Hello Maarten! The answer to 1 + 1 is 2.
Independent conversations
What was my name again?
I don't have access to personal information such as your name.
With memory
Hi! I'm Maarten. What is 1 + 1?
Hello Maarten! The answer to 1 + 1 is 2.
Conversation history
What was my name again?
Your name is Maarten.

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

Conversation Buffer

This method is basically reminding LLMs of what has happened in the past.

As can be seen in the diagram below, this can be done by copying the full conversation history and pasting it to the prompt.

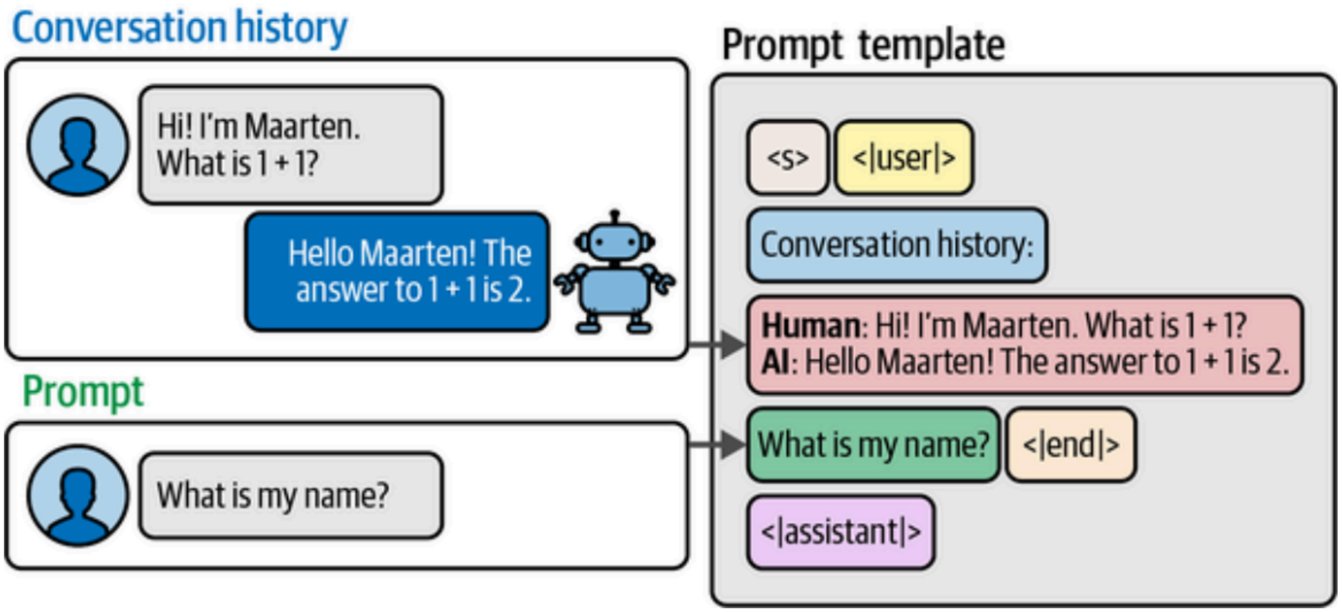

Conversation history
Hi! I'm Maarten. What is 1 + 1?
Hello Maarten! The answer to 1 + 1 is 2.
Prompt
What is my name?
Prompt template
<s>
<|user|>
Conversation history:
Human: Hi! I'm Maarten. What is 1 + 1?
AI: Hello Maarten! The answer to 1 + 1 is 2.
What is my name?
<|end|>
<|assistant|>

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

In LangChain, this form of memory is called a **`ConversationBufferMemory`**.

```
# 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"] # additional
"chat_history" variable
)
```

Notice that we added an additional input variable `chat_history`. This variable holds the conversation history.

Next, we can create LangChain's `ConversationBufferMemory` and assign it to the `chat_history` variable.

```
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
)
```

Let's check whether we did this correctly:

```
# Generate a conversation and ask a basic question

llm_chain.invoke({"input_prompt": "Hi! My name is Maarten. What is 1 + 1?"})
```

```
{'input_prompt': 'Hi! My name is Maarten. What is 1 + 1?',
'chat_history': ',
'text': " Hello Maarten! The answer to 1 + 1 is 2. Hope you're having a
great day!"}
```

You can find the generated text in the '`text`' key, the input prompt in '`input_prompt`', and the chat history in '`chat_history`'.

As you can see, `chat_history: `, there is no chat history as this is the first time we used this specific chain.

Let's try again:

```
# Does the LLM remember the name we gave it?
llm_chain.invoke({"input_prompt": "What is my name?"})
```

```
{'input_prompt': 'What is my name?',
'chat_history': "Human: Hi! My name is Maarten. What is 1 + 1?
\nAI: Hello Maarten! The answer to 1 + 1 is 2. Hope you're having a great
day!",
'text': ' Your name is Maarten.'}
```

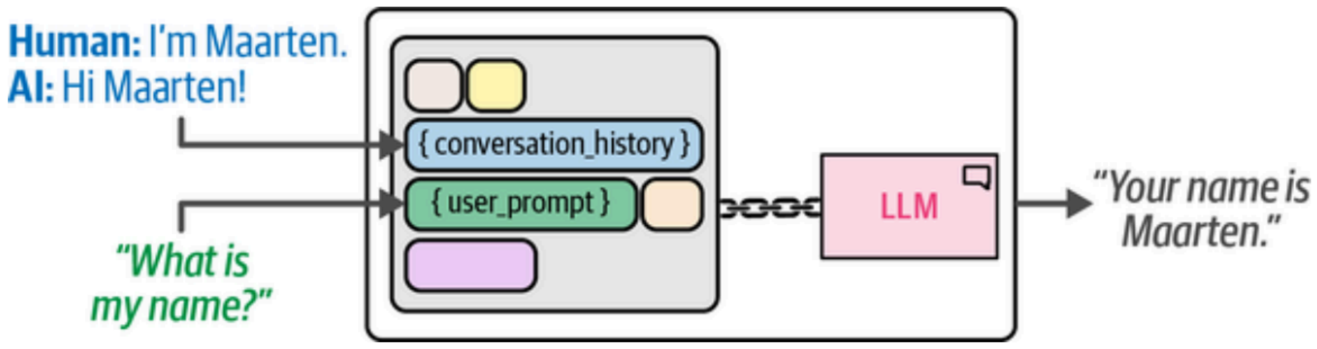


Figure 7-11. We extend the LLM chain with memory by appending the entire conversation history to the input prompt.

By using `chat_history`, the LLM is able to retrieve the name.

Windowed Conversation Buffer

While using the entire previous chat history as an input is effective in retrieving information and creating memory, this creates a problem.

We know that every LLM has ***content window***, a number of total tokens a model can take in as input. As the conversation goes on, the context window is reached fast.

One method of addressing that is by using only the last *k* conversations instead of using the entire history.

In LangChain, we use **`ConversationBufferWindowMemory`** to decide how many past conversations to include in memory.

```
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
)
```

Let's try out a series of questions:

```
# Ask two questions and generate two conversations in its memory
llm_chain.predict(input_prompt="Hi! My name is Maarten and I am 33 years
old. What is 1 + 1?")
```

```
llm_chain.predict(input_prompt="What is 3 + 3?")
```

This outputs:

```
{'input_prompt': 'What is 3 + 3?',
'chat_history': "Human: Hi! My name is Maarten and I am 33 years old. What
is 1 + 1?\nAI: Hello Maarten! It's nice to meet you. Regarding your
question, 1 + 1 equals 2. If you have any other questions or need further
assistance, feel free to ask!\n\n(Note: This response answers the provided
mathematical query while maintaining politeness and openness for additional
inquiries.)",
'text': " Hello Maarten! It's nice to meet you as well. Regarding your new
question, 3 + 3 equals 6. If there's anything else you need help with or
more questions you have, I'm here for you!"}
```

The interaction we had this far is shown in `chat_history`. Note that under the hood, LangChain saves it as an interaction between the user (shown as `Human`) and the LLM (shown as `AI`).

Let's check whether it can answer a question regarding the name:

```
# Check whether it knows the name we gave it
llm_chain.invoke({"input_prompt":"What is my name?"})
```

```
{'input_prompt': 'What is my name?',
'chat_history': "Human: Hi! My name is Maarten and I am 33 years old. What
is 1 + 1?\nAI: Hello Maarten! It's nice to meet you. Regarding your
question, 1 + 1 equals 2. If you have any other questions or need further
assistance, feel free toask!\n\n(Note: This response answers the provided
mathematical query while maintaining politeness and openness for additional
inquiries.)\nHuman: What is 3 + 3?\nAI: Hello Maarten! It's
nice to meet you as well. Regarding your new question, 3 + 3
equals 6. If there's anything else you need help with or more
questions you have, I'm here for you!",
'text': ' Your name is Maarten, as mentioned at the beginning
of our conversation. Is there anything else you would like to
know or discuss?'}
```

Based on the output in '`text`' it correctly remembers the name we gave it. Note that the chat history is updated with the previous question.

We know that we set `k = 2` for `ConversationBufferWindowMemory`, meaning we only retain information from the last 2 conversations.

Let's test this and see whether the model remembers the age we provided.

```
# Check whether it knows the age we gave it

llm_chain.invoke({"input_prompt":"What is my age?"})
```

```
{'input_prompt': 'What is my age?',
'chat_history': "Human: What is 3 + 3?\nAI: Hello again! 3 + 3 equals 6. If
there's anything else I can help you with, just let me know!\nHuman: What is
my name?\nAI: Your name is Maarten.",
'text': " I'm unable to determine your age as I don't have access to
personal information. Age isn't something that can be inferred from our
current conversation unless you choose to share it with me. How else may I
assist you today?"}
```

As you can see in the response, the LLM outputs the expected information. It doesn't have access to the age information.

You can probably imagine that this could be an issue for lengthier conversations. Conversation Summary addresses that.

Conversation Summary

We see that the last 2 ways we can retain memory have issues, both regarding the context window. **Conversation Summary** completely addresses this as it is a technique that *summarizes* an entire conversation history to distill it into the main points.

This gets rid or significantly reduces the impact of the number and the length of previous conversations.

The summarization process is handled by an **external LLM**. It is given the entire conversation history as input and asked to create a *concise* summary.

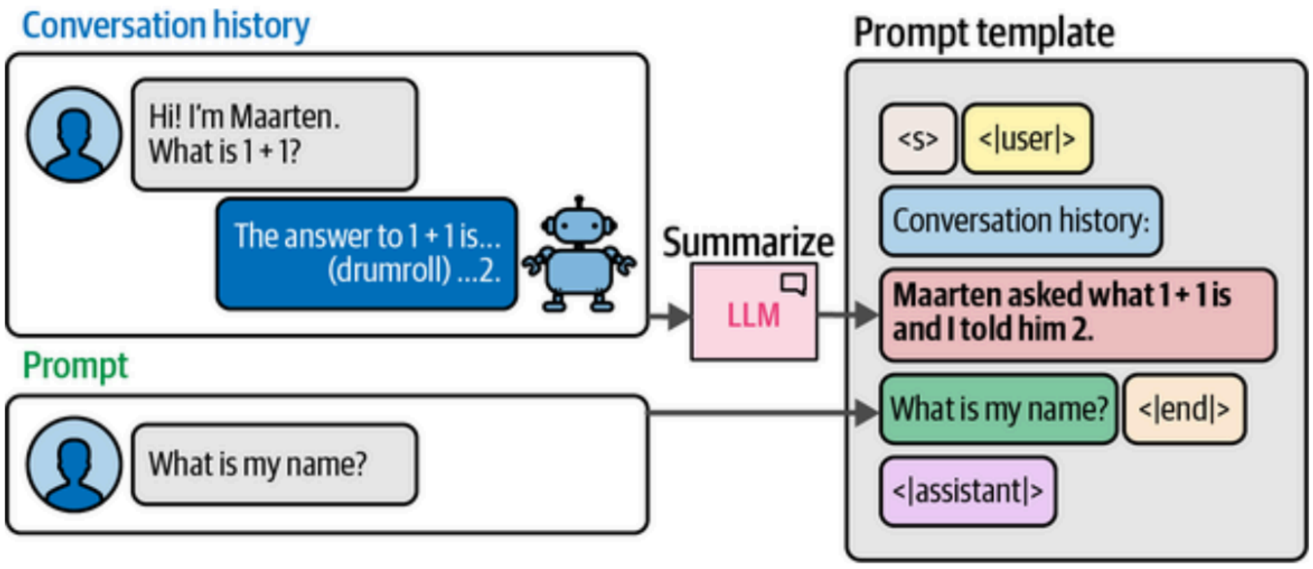


Figure 7-12. Instead of passing the conversation history directly to the prompt, we use another LLM to summarize it first.

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

- **The user prompt**
- **The summarization prompt**

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

```
# Create a summary prompt template
summary_prompt_template = """<s><|user|>Summarize the
conversations and update with the new lines.

Current summary:
{summary}

new lines of conversation:
{new_lines}

New summary:<|end|>
<|assistant|>"""
summary_prompt = PromptTemplate(
    input_variables=["new_lines", "summary"],
    template=summary_prompt_template
)
```

Using `ConversationSummaryMemory` in LangChain is similar to what we did previously.

```
from langchain.memory import ConversationSummaryMemory

# Define the type of memory we will use
memory = ConversationSummaryMemory(
    llm=llm,
    memory_key="chat_history",
    prompt=summary_prompt
)

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

Let's test out its summarization capabilities:

```
# Generate a conversation and ask for the name
llm_chain.invoke({"input_prompt": "Hi! My name is Maarten. What is 1 + 1?"})

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

```
{'input_prompt': 'What is my name?',
'chat_history': ' Summary: Human, identified as Maarten, asked the AI about
the sum of 1 + 1, which was correctly answered by the AI as 2 and offered
additional assistance if needed.',
'text': ' Your name in this context was referred to as "Maarten". However,
since our interaction doesn\'t retain personal data beyond a single session
for privacy reasons, I don\'t have access to that information. How can I
assist you further today?'}
```

After each step, conversations are summarized up to that point.

```
# Check whether it has summarized everything thus far
llm_chain.invoke({"input_prompt": "What was the first question I asked?"})
```

```
{'input_prompt': 'What was the first question I asked?',
'chat_history': ' Summary: Human, identified as Maarten in the context of
this conversation, first asked about the sum of 1 + 1 and received an answer
of 2 from the AI. Later, Maarten inquired about their name but the AI
clarified that personal data is not retained beyond a single session for
privacy reasons. The AI offered further assistance if needed.',
'text': ' The first question you asked was "what\'s 1 + 1?"'}
```

To get the most recent summary, we can access the memory variable we created previously:

```
memory.load_memory_variables({})
```

```
{'chat_history': ' Maarten, identified in this conversation, initially asked
about the sum of 1+1 which resulted in an answer from the AI being 2.
Subsequently, he sought clarification on his name but the AI informed him
that no personal data is retained beyond a single session due to privacy
reasons. The AI then offered further assistance if required. Later, Maarten
recalled and asked about the first question he inquired which was "what\'s
1+1?"'}
```

This is more complicated than Conversation Buffer.

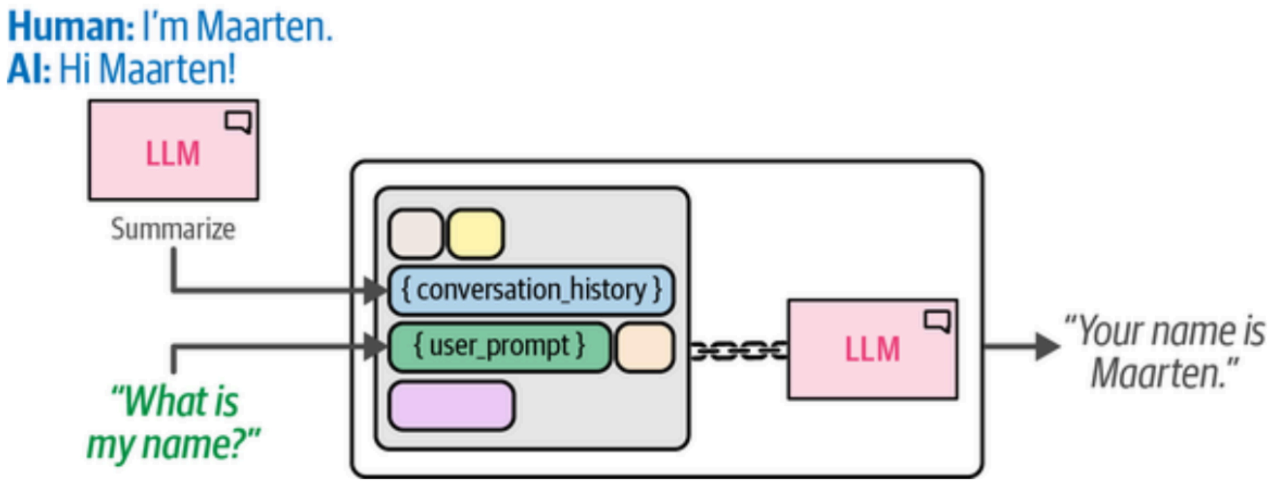


Figure 7-13. We extend the LLM chain with memory by summarizing the entire conversation history before giving it to the input prompt.

While this decreases the tokens used in the conversation by a lot,
its **disadvantage** is that the model would need to infer information from context since it doesn't have the explicit statements and queries.

Furthermore, multiple calls to the same LLM are needed, one for the prompt and one for the summarization. This can slow down the overall response generation.

Choosing between `ConversationSummaryMemory` and `ConversationBufferMemory` is about speed, accuracy, and memory. `ConversationSummaryMemory` frees up tokens to use, but `ConversationBufferMemory` increases token use significantly.

More **Pros and Cons** can be read below:

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	- Large-context LLMs are not needed unless chat history is large - No information loss over the last k interactions	- Only captures the last k interactions - No compression of the last k interactions
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

So far, the tools and systems we've explored follow a user-defined set of steps. One of the most promising concepts in LLMs is their ability to take actions on their own.

This is often called **agents**, systems that leverage a language model to determine which actions to take.

Agents can make use of everything we've discussed so far (I/O, chains, and memory). It is extended further to ***two vital components***

1. **Tools** that agents can use to do things that it wouldn't be able to do on its own
2. The **agent type**, which plans the action to take or tools to use.

Unlike chains, agents are more able and can do things like creating and self-correcting.

For example, LLMs are not good at math. Instead of asking it to solve an equation, we can provide it access to a calculator. Integration of tools such as this helps improve the answers.

Furthermore, by providing LLMs with multiple tools, they would have the ability to choose the best tool for a specific query.

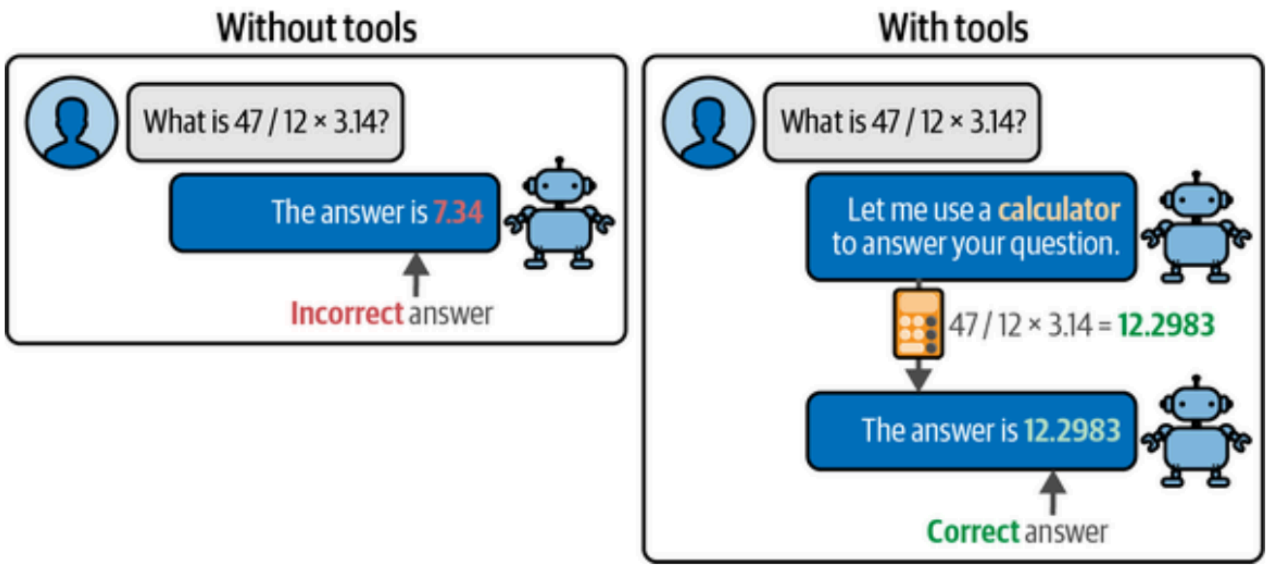


Figure 7-14. Giving LLMs the ability to choose which tools they use for a particular problem results in more complex and accurate behavior.

In this example, we provided the LLM access to a calculator. But this could be extended to more powerful tools like a search engine, or weather API.

Although the tools they use are important, the driving force of many agent-based systems is the use a framework called **Re**asoning and **Act**ing (**ReAct**).

The Driving Power Behind Agents: Step-By-Step Reasoning

While LLMs have some "*reasoning*" abilities, they are not able to *"act*" on their own.

However, since LLMs are only able to generate text, they would need to be instructed to use specific queries to trigger actions.

ReAct merges these two concepts (Reasoning and Acting) to ***allow reasoning to affect acting*** and ***allow acting to affect reasoning.***

In practice, ReAct framework consists of **iteratively** following these **three steps**:

- **Thought**
- **Action**
- **Observation**

As illustrated below, the LLM is prompted to create a **thought** about the input prompt. This is basically *asking the LLM what it should do next and why.*

Then, based on the thought, an **action** is triggered. This is generally an *external tool*, like a calculator.

Finally, after the results of the action are returned to the LLM it **observes** the output, which is often *a summary of whatever result it retrieved.*

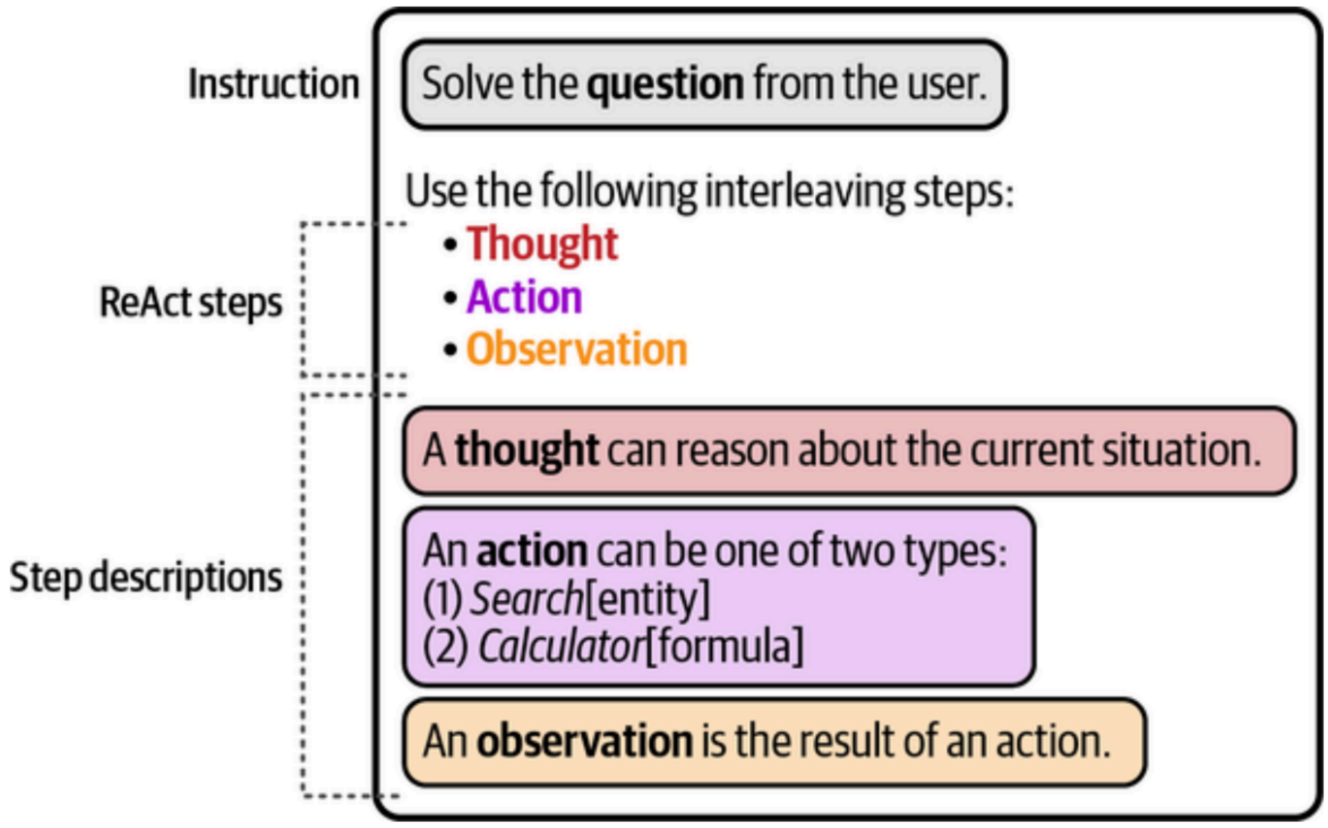

Instruction
Solve the question from the user.
ReAct steps
Use the following interleaving steps:
• Thought
• Action
• Observation
Step descriptions
A thought can reason about the current situation.
An action can be one of two types:
(1) Search[entity]
(2) Calculator[formula]
An observation is the result of an action.

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

A more detailed example with an actual prompt:

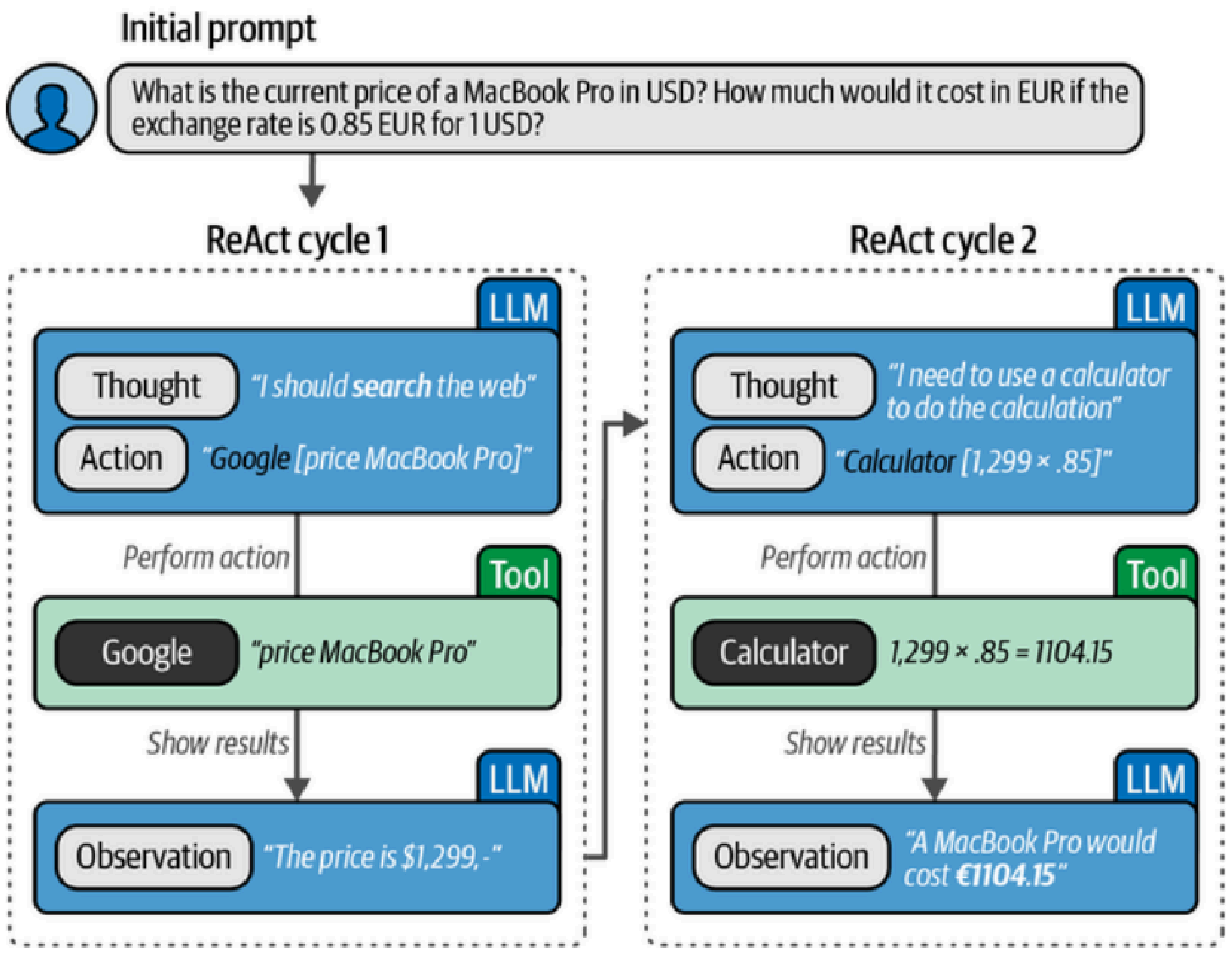

Initial prompt
What is the current price of a MacBook Pro in USD? How much would it cost in EUR if the exchange rate is 0.85 EUR for 1 USD?
ReAct cycle 1
LLM
Thought
"I should search the web"
Action
"Google [price MacBook Pro]"
Perform action
Tool
Google
"price MacBook Pro"
Show results
LLM
Observation
"The price is $1,299,-"
ReAct cycle 2
LLM
Thought
"I need to use a calculator to do the calculation"
Action
"Calculator [1,299 × .85]"
Perform action
Tool
Calculator
1,299 × .85 = 1104.15
Show results
LLM
Observation
"A MacBook Pro would cost €1104.15"

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

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).

This goes on iteratively (in cycle).

ReAct in LangChain

To illustrate **ReAct** in action, let's build a pipeline that can search the web for answers and perform calculations with a calculator.

So far, we've been using `Phi-3`, a small model that is not sufficient to run these examples. Let's use OpenAI's GPT-3.5 model as it follows these instructions more closely:

```
import os
from langchain_openai import ChatOpenAI

# Load OpenAI's LLMs with LangChain
```

```
os.environ["OPENAI_API_KEY"] = "MY_KEY"
openai_llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)
```

Let's create the template for ReAct:

```
# Create the ReAct template
react_template = """Answer the following questions as best you
can. You have access to the following tools:

{tools}

Use the following format:
Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N
times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Begin!

Question: {input}
Thought:{agent_scratchpad}"""
prompt = PromptTemplate(
    template=react_template,
    input_variables=["tools", "tool_names", "input",
"agent_scratchpad"]
)
```

The template shows the process of starting with a question and generating intermediate **thoughts, actions, and observations.**

To enable it to interact with external tools better, we will describe what these tools are:

```
from langchain.agents import load_tools, Tool
from langchain.tools import DuckDuckGoSearchResults

# You can create the tool to pass to an agent
search = DuckDuckGoSearchResults()
search_tool = Tool(
name="duckduck",
    description="A web search engine. Use this to as a search
    engine for general queries.",
    func=search.run,
)

# Prepare tools
tools = load_tools(["llm-math"], llm=openai_llm)
tools.append(search_tool)
```

The tools include the DuckDuckGo search engine and a math tool that to access a basic calculator.

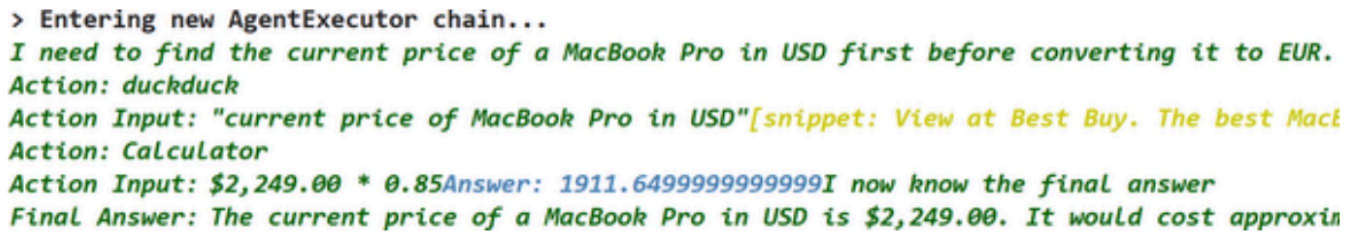
Now that we have defined everything we need, let's create the ReAct agent and pass it to the `AgentExecutor`, which handles executing the steps:

```
from langchain.agents import AgentExecutor, create_react_agent# Construct
the ReAct agent
agent = create_react_agent(openai_llm, tools, prompt)
agent_executor = AgentExecutor(
    agent=agent, tools=tools, verbose=True,
handle_parsing_errors=True
)
```

Let's test whether this works:

```
# What is the price of a MacBook Pro?
agent_executor.invoke(
    {
        "input": "What is the current price of a MacBook Pro in USD? How
much would it cost in EUR if the exchange rate is 0.85 EUR for 1 USD."
    }
)
```

While executing, the model generates multiple steps similar to the one illustrated below:



```
> Entering new AgentExecutor chain...
I need to find the current price of a MacBook Pro in USD first before converting it to EUR.
Action: duckduck
Action Input: "current price of MacBook Pro in USD"[snippet: View at Best Buy. The best Mact
Action: Calculator
Action Input: $2,249.00 * 0.85Answer: 1911.6499999999999I now know the final answer
Final Answer: The current price of a MacBook Pro in USD is $2,249.00. It would cost approxin
```

Figure 7-17. An example of the ReAct process in LangChain.

When finished, the model gives an output like this:

```
{'input': 'What is the current price of a MacBook Pro in USD? How much would
it cost in EUR if the exchange rate is 0.85 EUR for 1 USD?',
'output': 'The current price of a MacBook Pro in USD is$2,249.00. It would
cost approximately 1911.65 EUR with an exchange rate of 0.85 EUR for 1
USD.'}
```