Chapter 6. Autonomous Agents with Memory and Tools

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the author at ccollins@oreilly.com.

This chapter dives deeper into the importance of chain-of-thought reasoning and the ability of large language models (LLMs) to reason through complex problems as agents. By breaking down complex problems into smaller, more manageable components, LLMs can provide more thorough and effective solutions. You will also learn about the components that make up autonomous agents, such as inputs, goal or reward functions, and available actions.

Chain-of-Thought

The ability of AI to reason through complex problems is essential for creating effective, reliable, and user-friendly applications.

Chain-of-thought reasoning (CoT) is a method of guiding LLMs through a series of steps or logical connections to reach a conclusion or solve a problem. This approach is particularly useful for tasks that require a deeper understanding of context or multiple factors to consider.

CoT is asking an LLM to *think* through complex problems, breaking them down into smaller, more manageable components. This allows the LLM to focus on each part individually, ensuring a more thorough understanding of the issue at hand.

In practice, chain-of-thought reasoning might involve:

- Asking an LLM to provide explanations for its decisions.
- Planning multiple steps before deciding on a final answer.

In the following sections, you'll explore examples of both ineffective and effective chain-of-thought reasoning. We will also discuss various techniques for building effective chain-of-thought reasoning and how these can be integrated into AI applications.

Let's imagine that a user wants the AI to generate a comprehensive marketing plan for promoting a new software product.

Input:

```
Create a marketing plan for a new software product.
```

Output:

```
1. Launch social media campaign
2. Send out email newsletters
3. Offer limited-time discounts
4. Collaborate with influencers
5. Organize webinars
```

In that example GPT-4 doesn't use chain-of-thought reasoning and it does not address the specific aspects of the marketing plan. The LLM generates a generic list of marketing strategies that could apply to any product, rather than focusing on the unique characteristics of the new software product.

Input:

```
Create a detailed marketing plan for a new project management software
product targeting small and medium-sized businesses.

The budget is $20,000, and we want to focus on digital marketing channels.

Consider the success of similar products in the market and provide a
step-by-step plan that includes the most effective promotional tactics.
```

Output:

```
Market Research
a. Analyze competitors and their marketing strategies
b. Identify the unique selling points of the new project management software
c. Define the target audience and their pain points

Branding and Messaging
a. Develop a consistent brand identity
b. Craft compelling messaging that addresses the target audience's pain
points
c. Create marketing assets such as website, blog, and social media content

...

Email Marketing
a. Develop an email marketing strategy to nurture leads and convert them
into customers
b. Send personalized onboarding emails to new users with helpful tips and
resources
```

Now GPT-4 specifically addresses the unique characteristics of the new software product, demonstrating effective chain-of-thought reasoning.

GIVE DIRECTION

Take note of the phrase *step by step*, a critical element in CoT. By incorporating this phrase into your prompt, you're asking the LLM to reason through the steps that are required to generate a highly effective software product.

Also, by providing a $20,000 budget and the type of software, GPT-4 is able to provide a much more relevant and contextualized response.

Agents

Generative AI models have given rise to an *agent-based architecture*. Conceptually, an agent acts, perceives, and makes decisions within a specified environment to achieve predefined objectives.

Agents can take various actions such as executing a Python function, afterwards the agent will observe what happens and will decide on whether it is finished or what action to take next.

The agent will continously loop through a series of actions, observations until there are no further actions, as you can see in the following pseudocode:

```
next_action = agent.get_action(...)
while next_action != AgentFinish:
    observation = run(next_action)
    next_action = agent.get_action(..., next_action, observation)
return next_action
```

The behavior of the agent is governed by three principal components:

1. *Inputs*: These are the sensory stimuli or data points the agent receives from its environment. Inputs can be diverse, ranging from visual (like images) and auditory (like audio files) to thermal signals and beyond.
2. *Goal or reward function*: This represents the guiding principle for an agent's actions. In goal-based frameworks, the agent is tasked with reaching a specific end state. In a reward-based setting, the agent is driven to maximize cumulative rewards over time, often in dynamic environments.
3. *Available actions*: The *action space* is the range of permissible actions an agent can undertake at any given moment. The breadth and nature of this space are contingent upon the task at hand.

To explain these concepts further, consider a self-driving car:

- *Inputs*: The car's sensors, such as cameras, LIDAR, and ultrasonic sensors, provide a continuous stream of data about the environment. This can include information about nearby vehicles, pedestrians, road conditions, and traffic signals.
- *Goal or reward function*: The primary goal for a self-driving car is safe and efficient navigation from point A to point B. If we were to use a reward-based system, the car might receive positive rewards for maintaining a safe distance from other objects, adhering to speed limits, and following traffic rules. Conversely, it could receive negative rewards for risky behaviors, like hard braking or veering off the lane. Tesla specifically uses miles driven without an intervention as their reward function.
- *Available actions*: The car's action space includes accelerating, decelerating, turning, changing lanes, and more. Each action is chosen based on the current input data and the objective defined by the goal or reward function.

You'll find that agents in systems like self-driving cars rely on foundational principles like inputs, goal/reward functions, and available actions.

However, when delving into the realm of LLMs like GPT, there's a bespoke set of dynamics that cater specifically to their unique nature.

Here's how they align with your needs:

- *Inputs*: For LLMs, the gateway is primarily through text. But that doesn't restrain the wealth of information you can use. Whether you're dealing with thermal readings, musical notations, or intricate data structures, your challenge lies in molding these into textual representations suitable for an LLM. Think about videos: while raw footage might seem incompatible, video text transcriptions allow an LLM to extract insights for you.
- *Harnessing goal-driven directives*: LLMs primarily use goals defined within your text prompts. By creating effective prompts with objectives, you're not just accessing the LLMs vast knowledge; you're effectively charting its reasoning path. Think of it as laying down a blueprint: your specific prompt instructs the model, guiding it to dissect your overarching objective into a systematic sequence of steps.
- *Crafting action through functional Tools*: LLMs are not limited to mere text generation; there's so much more you can achieve. By integrating *ready-made tools* or *custom developed tools*, you can equip LLMs to undertake diverse tasks, from API calls to database engagements or even orchestrating external systems. Tools can be written in any programming language, and by adding more tools you are effectively *expanding the action space* of what an LLM can achieve.

There are also different components that are directly applicable to LLMs:

- *Memory*: It's ideal to store state between agent steps, this is particularly useful for chatbots where remembering the previous chat history provides a better user experience.
- *Agent planning/execution strategies*: There are multiple ways to achieve a high level goal, of which a mixture of planning and executing are essential.
- *Retrieval*: LLMs can use different types of retrieval methods. Semantic similarity within vector databases is the most common, but there are others such as including custom information from a SQL database into prompts.

Let's dive deeper into the shared and different components and explore the implementation details.

Reason and Act (reAct)

There are many agent frameworks that ultimately aim to improve LLM responses towards a goal. The original framework was *reAct*, which is an improved version of CoT, allowing an LLM to create observations after taking actions via tools. These observations are then turned into *thoughts* about what would be the *right tool* to use within the next step (Figure 6-1). The LLM continues to reason until either a `'Final Answer'` string value is present or a maximum number of iterations has taken place.

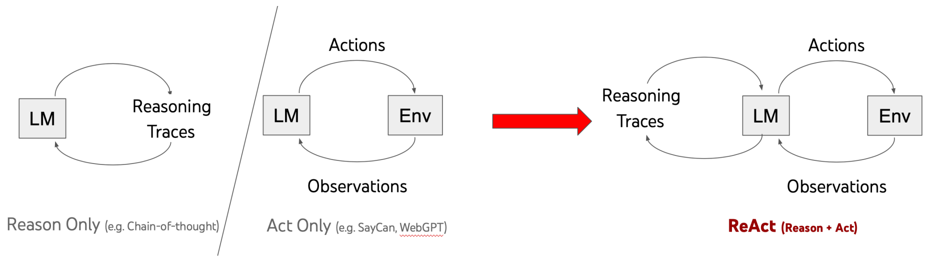

Figure 6-1. The ReAct Framework

The ReAct framework uses a mixture of task decomposition, a thought loop, and multiple tools to solve questions. Let's explore the thought loop within reAct:

1. Observe the environment.
2. Interpret the environment with a thought.
3. Decide on an action.
4. Act on the environment.
5. Repeat steps 1 - 4 until you find a solution or you've done too many iterations (the solution is "I've found the answer").

You can easily create a reAct style prompt by using the preceding thought loop while also providing the LLM with several inputs such as:

- `{question}`: The query that you want answered.
- `{tools}`: These refer to functions which can be used to accomplish a step within the overall task. It is common practice to include a list of tools, where each tool is a Python function, a name and description of the function and its purpose.

The following is a prompt that implements the reAct pattern with prompt variables wrapped in `{}` characters such as `{question}`:

```
You will attempt to solve the problem of finding the answer to a question.
Use chain-of-thought reasoning to solve through the problem, using the
following pattern:
```

```
1. Observe the original question:
original_question: original_problem_text
2. Create an observation with the following pattern:
observation: observation_text
3. Create a thought based on the observation with the following pattern:
thought: thought_text
4. Use tools to act on the thought with the following pattern:
action: tool_name
action_input: tool_input

Do not guess or assume the tool results. Instead, provide a structured
output that includes the action and action_input.

You have access to the following tools: {tools}.

original_problem: {question}

Based on the provided tool result:

Either provide the next observation, action, action_input, or the final
answer if available.

If you are providing the final answer, you must return the following pattern:
"I've found the answer: final_answer"
```

- The introduction of the prompt clearly establishes the LLMs purpose: `You will attempt to solve the problem of finding the answer to a question.`
- The problem-solving approach is then outlined: `Use chain-of-thought reasoning to solve through the problem, using the following pattern:`
- The steps in the chain-of-thought reasoning are then laid out:
  - The LLM starts by observing the original question and subsequently formulates an observation about it: `original_question: original_problem_text`, `observation: observation_text`.
  - Based on this observation, the AI should formulate a thought that signifies a step in the reasoning process: `thought: thought_text`.
  - Having established a thought, it then decides on an action using one of the available tools: `action: tool_name`, `action_input: tool_input`.
- The LLM is then reminded not to make assumptions about what a tool might return, and that it should explicitly outline its intended action and the corresponding input.
- `You have access to the following tools: {tools}` This line communicates to the LLM what tools it has available for solving

the problem.

- The actual problem that the LLM must solve is then introduced: `original_problem: {question}`.
- Finally, instructions are provided on how the LLM should respond based on the results of its actions. It can either continue with new observations, actions, and inputs, or if a solution is found, provide the final answer.

The prompt outlines a systematic problem-solving process in which the LLM observes a problem, thinks about it, decides on an action, and repeats this process until a solution is discovered.

Reason and Act Implementation

Now that you're aware of reAct, it's important to create a simple Python implementation that replicates what LangChain does automatically, allowing you to build the intuition about what's truly happening between the LLM responses.

To keep it simple, this example will not implement looping and will assume that the output can be obtained from a single tool call.

To create a basic reAct implementation, you'll implement the following:

1. At every thought, you need to extract the tool that the LLM wants to use. Therefore you'll extract the last `action` and `action_input`. The `action` represents the tool name, while the `action_input` consists of the values of the function arguments.
2. Check whether the LLM thinks that it has found the final answer, in which case the thought loop has ended.

You can use regular expressions to extract the `action` and `action_input` values from the LLM response:

```
import re

# Sample text
text = """
Action: search_on_google
Action_Input: Tom Hanks current wife

action: search_on_wikipedia
action_input: How old is Rita Wilson in 2023

action : search_on_google
action input: some other query
```

```
"""

# Compile regex patterns
action_pattern = re.compile(r"(?i)action\s*:\s*([^\n]+)", re.MULTILINE)
action_input_pattern = re.compile(r"(?i)action\s*_*input\s*:\s*([^\n]+)",
re.MULTILINE)

# Find all occurrences of action and action_input
actions = action_pattern.findall(text)
action_inputs = action_input_pattern.findall(text)

# Extract the last occurrence of action and action_input
last_action = actions[-1] if actions else None
last_action_input = action_inputs[-1] if action_inputs else None

print("Last Action:", last_action)
print("Last Action Input:", last_action_input)
# Last Action: search_on_google
# Last Action Input: some other query
```

Let's break down the regular expression to extract the `action`:

- `action_pattern = re.compile(r"(?i)action\s*:\s*([^\n]+)", re.MULTILINE)`
- `(?i)`: This is called an inline flag and makes the regex pattern case-insensitive. It means that the pattern will match "action", "Action", "ACTION", or any other combination of uppercase and lowercase letters.
- `action`: This part of the pattern matches the word "action" literally. Due to the case-insensitive flag, it will match any capitalization of the word.
- `\s*`: This part of the pattern matches zero or more whitespace characters (spaces, tabs, etc.). The `\*` means *zero or more* and `\s` is the regex shorthand for a whitespace character.
- `:` This part of the pattern matches the colon character literally.
- `\s*`: This is the same as the previous `\s\*` part, matching zero or more whitespace characters after the colon.
- `+([^\n]++)`: This pattern is a capturing group, denoted by the parentheses. It matches one or more characters that are *not a newline character*. The `^` inside the square brackets `[]` negates the character class, and `\n` represents the newline character. The `+` means *one or more*. The text matched by this group will be extracted when using the `findall()` function.
- `re.MULTILINE`: This is a flag passed to `re.compile()` function. It tells the regex engine that the input text may have multiple lines, so the pattern should be applied line by line.

- In regular expressions, square brackets `[]` are used to define a character class, which is a set of characters that you want to match. For example, `[abc]` would match any single character that is either `'a'`, `'b'`, or `'c'`.
- When you add a caret `(^)` at the beginning of the character class, it negates the character class, meaning it will match any character that is *not in the character class*. In other words, it inverts the set of characters you want to match.
- So, when we use `[^abc]`, it will match any single character that is *not* `'a'`, `'b'`, or `'c'`. In the regex pattern `+([^\n]++)`, the character class is `[^n]`, which means it will match any character that is *not* a newline character (`\n`). The `+` after the negated character class means that the pattern should match one or more characters that are not newlines.
- By using the negated character class `[^n]` in the capturing group, we ensure that the regex engine captures text up to the end of the line without including the newline character itself. This is useful when we want to extract the text after the word "action" or "action input" up to the end of the line.

Overall, this regular expression pattern matches the word "action" (case-insensitive) followed by optional whitespace, a colon, optional whitespace again, and then captures any text up to the end of the line.

The only difference between these two regex patterns is the literal text they are looking for at the beginning:

1. `action_pattern` looks for the word `"action"`.
2. `action_input_pattern` looks for the word `"action_input"`.

You can now abstract the regex into a Python function that will always find the last `action` and `action_input`:

```
def extract_last_action_and_input(text):
    # Compile regex patterns
    action_pattern = re.compile(r"(?i)action\s*:\s*([^\n]+)", re.MULTILINE)
    action_input_pattern = re.compile(
        r"(?i)action\s*_*input\s*:\s*([^\n]+)", re.MULTILINE
    )

    # Find all occurrences of action and action_input
    actions = action_pattern.findall(text)
    action_inputs = action_input_pattern.findall(text)

    # Extract the last occurrence of action and action_input
    last_action = actions[-1] if actions else None
```

```
    last_action_input = action_inputs[-1] if action_inputs else None

    return {"action": last_action, "action_input": last_action_input}

extract_last_action_and_input(text)
# {'action': 'search_on_google', 'action_input': 'some other query'}
```

To determine and extract whether LLM has discovered the final answer you can also use regular expressions:

```
def extract_final_answer(text):
    final_answer_pattern = re.compile(
        r"(?i)I've found the answer:\s*([^\n]+)", re.MULTILINE
    )
    final_answers = final_answer_pattern.findall(text)
    if final_answers:
        return final_answers[0]
    else:
        return None

final_answer_text = "I've found the answer: final_answer"
print(extract_final_answer(final_answer_text))
# final_answer
```

WARNING

LLMs do not always respond in the intended way, so your application needs to be able to handle regex parsing errors. Several approaches include using an LLM to fix the previous LLM response, or making another new LLM request with the previous state.

You can now combine all of the components together, and here is a step-by-step explanation:

```
from langchain_openai.chat_models import ChatOpenAI
from langchain.prompts.chat import SystemMessagePromptTemplate
```

Initialize the `ChatOpenAI` instance:

```
chat = ChatOpenAI(model_kwargs={"stop": ["tool_result:"],})
```

Adding a `stop` sequence forces an LLM to stop generating new tokens after encounting the phrase `"tool_result:"`. This helps by stopping hallucinations for tool usage.

Define the available tools:

```
tools = {}

def search_on_google(query: str):
    return f"Jason Derulo doesn't have a wife or partner."

tools["search_on_google"] = {
    "function": search_on_google,
    "description": "Searches on google for a query",
}
```

Set the base prompt template:

```
base_prompt = """
You will attempt to solve the problem of finding the answer to a question.
Use chain-of-thought reasoning to solve through the problem, using the
following pattern:

1. Observe the original question:
original_question: original_problem_text
2. Create an observation with the following pattern:
observation: observation_text
3. Create a thought based on the observation with the following pattern:
thought: thought_text
4. Use tools to act on the thought with the following pattern:
action: tool_name
action_input: tool_input

Do not guess or assume the tool results. Instead, provide a structured
output that includes the action and action_input.

You have access to the following tools: {tools}.

original_problem: {question}
"""
```

Generate the model output:

```
output = chat.invoke(SystemMessagePromptTemplate \
.from_template(template=base_prompt) \
.format_messages(tools=tools, question="Is Jason Derulo with a partner?"))
print(output)
```

Extract the last `action`, `action_input`, and call the relevant function:

```
tool_name = extract_last_action_and_input(output.content)["action"]
tool_input = extract_last_action_and_input(output.content)["action_input"]
tool_result = tools[tool_name]["function"](tool_input)
```

Print the tool details:

```
print(f"""The agent has opted to use the following tool:
tool_name: {tool_name}
tool_input: {tool_input}
tool_result: {tool_result}"""
)
```

Set the current prompt with the tool result:

```
current_prompt = """
Based on the provided tool result:
tool_result: {tool_result}

Either provide the next observation, action, action_input, or the final
answer if available. If you are providing the final answer, you must return
the following pattern: "I've found the answer: final_answer"
"""
```

Generate the model output for the current prompt:

```
output = chat.invoke(SystemMessagePromptTemplate. \
from_template(template=current_prompt) \
.format_messages(tool_result=tool_result))
```

Print the model output for the current prompt:

```
print("----------\n\nThe model output is:", output.content)
final_answer = extract_final_answer(output.content)
if final_answer:
    print(f"answer: {final_answer}")
else:
    print("No final answer found.")
```

Output:

```
'''content='1. Observe the original question:\nIs Jason Derulo with a
partner?\n\n2. Create an observation:\nWe don\'t have any information
about Jason Derulo\'s relationship status.\n\n3. Create a thought based
```

```
on the observation:\nWe can search for recent news or interviews to find
out if Jason Derulo is currently with a partner.\n\n4. Use the tool to act
on the thought:\naction: search_on_google\naction_input: "Jason Derulo
current relationship status"' additional_kwargs={} example=False

----------
The agent has opted to use the following tool:
tool_name: search_on_google
tool_input: "Jason Derulo current relationship status"
tool_result: Jason Derulo doesn't have a wife or partner.
----------

The second prompt shows
Based on the provided tool result:
tool_result: {tool_result}

Either provide the next observation, action, action_input, or the final
answer if available. If you are providing the final answer, you must
return the following pattern: "I've found the answer: final_answer"
----------

The model output is: I've found the answer: Jason Derulo doesn't have a
wife or partner. answer: Jason Derulo doesn't have a wife or partner.'''
```

The preceding steps provide a very simple reAct implementation. In this case the LLM decided to use the `search_on_google` tool with `"Jason Derulo current relationship status"` as the `action_input`.

NOTE

LangChain agents will automatically do all of the preceding steps in a concise manner, as well as provide multiple tool usage (through looping) and handling for tool failures when an agent can't parse the `action` or `action_input`.

Before exploring LangChain agents and what they have to offer, it's vital that you learn *tools* and how to create and use them.

Using Tools

As large language models such as GPT-4 can only generate text, providing tools that can perform other actions such as interacting with a database or reading/writing files provides an effective method to increase an LLM capabilities. A *tool is simply a pre-defined function* that permits the agent to take a specific action

A common part of an agent's prompt will likely include the following:

```
You are looking to accomplish: {goal}
You have access to the following {tools}
```

Most tools are written as functions within a programming language.

As you explore Langchain, you'll find that it offers three different approaches to tool creation/usage. You can either:

1. Create your own custom tools.
2. Use pre-existing tools.
3. Leverage `AgentToolkits`, which are multiple tools bundled together to accomplish a specific task.

Let's start by creating a custom tool that checks the length of a given string using LangChain:

```python
# Import necessary classes and functions:
from langchain.agents import AgentExecutor, create_react_agent
from langchain import hub
from langchain_openai import ChatOpenAI
from langchain.tools import Tool

# Defining the LLM to use:
model = ChatOpenAI()

# Function to count the number of characters in a string:
def count_characters_in_string(string):
    return len(string)

# Create a list of tools:
# Currently, only one tool is defined that counts characters in a text string.
tools = [
    Tool.from_function(
        func=count_characters_in_string,
        name="Count Characters in a text string",
        description="Count the number of characters in a text string",
    )
]

# Download a react prompt!
prompt = hub.pull("hwchase17/react")

# Construct the ReAct agent:
agent = create_react_agent(model, tools, prompt)

# Initialize an agent with the defined tools and
# Create an agent executor by passing in the agent and tools:
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
```

```
# Invoke the agent with a query to count the characters in the given word
agent_executor.invoke({"input": '''How many characters are in the word
"supercalifragilisticexpialidocious"?'''})

# 'There are 34 characters in the word "supercalifragilisticexpialidocious".'
```

Following the import of necessary modules, you initialize a `ChatOpenAI` chat model. Then create a function called `count_characters_in_string` that computes the length of any given string. This function is encapsulated within a `Tool` object, providing a descriptive name and explanation for its role.

Subsequently, you utilize `create_react_agent` to initialize your agent, combining the defined `Tool`, the `ChatOpenAI` model, and a react prompt pulled from the LangChain hub. This sets up a comprehensive interactive agent.

With `AgentExecutor`, the agent is equipped with the tools and verbose output is enabled, allowing for detailed logging.

Finally, `agent_executor.invoke` is executed with a query about the character count in "supercalifragilisticexpialidocious". The agent utilizes the defined tool to calculate and return the precise character count in the word.

In Example 6-1, you can see that the agent decided to use the `Action` called `Characters in a text string` with an `Action Input`: `'supercalifragilisticexpialidocious'`. This pattern is extremely familiar to the simplistic reAct implementation that you previously made.

Example 6-1. A single tool, agent output

```
Entering new AgentExecutor change...
I should count the number of characters in the word "supercalifragilisticexpilado
Action: Count Characters in a text string
Action Input: "supercalifragilisticexpiladocious"
Observation: 34
Thought: I now know the final answer
Final Answer: There are 34 characters in the word "supercalifragilisticexpiladoci
```

GIVING DIRECTION

Writing expressive names for your Python functions and tool descriptions will increase an LLMs ability to effectively choose the right tools.

Using LLMs as an API (OpenAI Functions)

As mentioned in Chapter 4, OpenAI released more fine-tuned LLMs tailored towards function calling. This is important because it offers an alternative against the standard reAct pattern for tool use. It's similar to reAct in that you're still utilizing an LLM as *a reasoning engine*.

As seen in Example 6-1, function calling allows an LLM to easily transform a user's input into a weather API call.

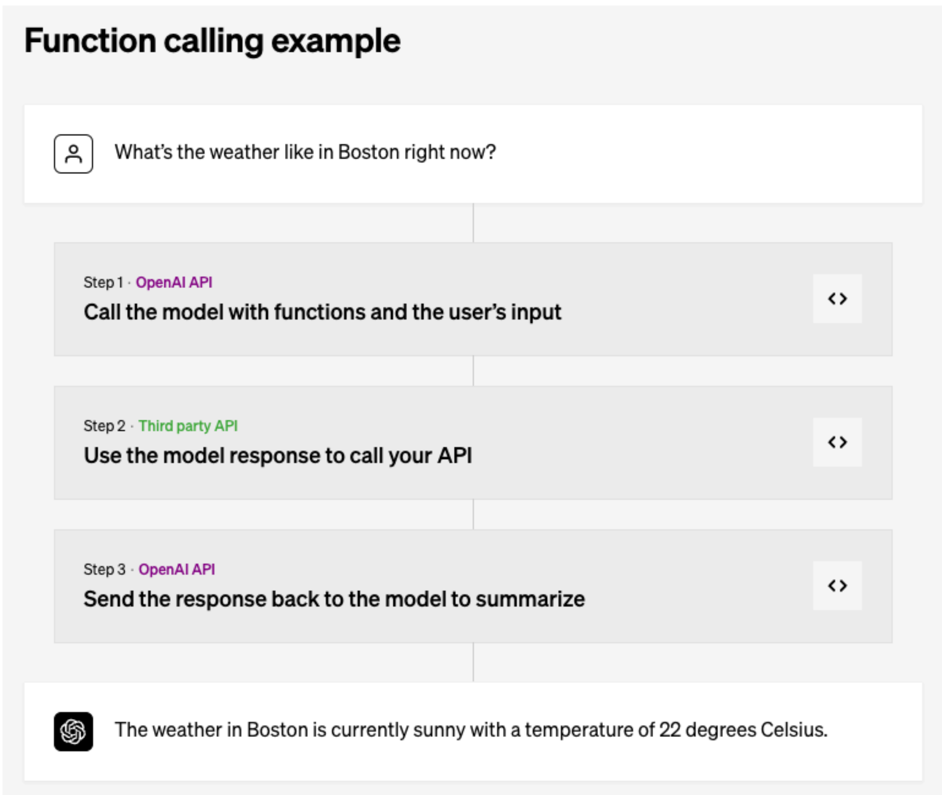

Figure 6-2. Function calling flow using OpenAI functions

LangChain allows users to effortlessly switch between different agent types including `reAct`, `OpenAI functions` and many more.

Refer to Table 6-1 for a comprehensive comparison of the different agent types:

Table 6-1. Comparison of agent types

Agent Type	Description
OpenAI Functions	Works with fine-tuned models like gpt-3.5-turbo-0613 and gpt-4-0613 for function calling. It intelligently outputs JSON objects for the function calls. Best for open source models and providers adopting this format. Note: Deprecated in favor of OpenAI Tools.
OpenAI Tools	Enhanced version for newer models, capable of invoking one or more functions. It intelligently outputs JSON objects for these function calls, optimizing the response efficiency and reducing response times in some architectures.
XML Agent	Ideal for language models like Anthropic's Claude, which excel in XML reasoning/writing. Best used with regular LLMs (not chat models) and unstructured tools accepting single string inputs.
JSON Chat Agent	Tailored for language models skilled in JSON formatting. This agent uses JSON to format its outputs, supporting Chat Models for scenarios requiring JSON outputs.
Structured Chat	Capable of using multi-input tools, this agent is designed for complex tasks requiring structured inputs and responses.
ReAct	Implements ReAct logic, using tools like TavilySearchResults for interactions with a document store or search tools.
Self-ask with Search	Utilizes the Intermediate Answer tool for factual question resolution, following the self-ask with search methodology. Best for scenarios requiring quick and accurate factual answers.

Let's use pre-packaged tools such as a `Calculator` to answer math questions using OpenAI function calling from the LangChain documentation:

```
# Import necessary modules and functions from the langchain package:
from langchain.chains import (
    LLMMathChain,
)
from langchain import hub
from langchain.agents import create_openai_functions_agent, Tool, AgentExecutor
from langchain_openai.chat_models import ChatOpenAI

# Initialize the ChatOpenAI with temperature set to 0:
model = ChatOpenAI(temperature=0)

# Create a LLMMathChain instance using the ChatOpenAI model:
llm_math_chain = LLMMathChain.from_llm(llm=model, verbose=True)
```

```python
# Download the prompt from the hub:
prompt = hub.pull("hwchase17/openai-functions-agent")

tools = [
    Tool(
        name="Calculator",
        func=llm_math_chain.run, # run the LLMMathChain
        description="useful for when you need to answer questions about math",
        return_direct=True,
    ),
]

# Create an agent using the ChatOpenAI model and the tools:
agent = create_openai_functions_agent(llm=model, tools=tools, prompt=prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

result = agent_executor.invoke({"input": "What is 5 + 5?"})
print(result)
# {'input': 'What is 5 + 5?', 'output': 'Answer: 10'}
```

After initiating the necessary libraries, you'll use `ChatOpenAI`, setting the `temperature` parameter to zero for deterministic outputs. By using `hub.pull("...")`, you can easily download prompts that have been saved on LangChainHub.

This model is then coupled with a tool named `Calculator` which leverages the capabilities of `LLMMathChain` to compute math queries. The OpenAI functions agent then decides to use the calculator tool to compute `5 + 5` and returns `Answer: 10`.

Following on, you can equip an agent with multiple tools, enhancing its versatility. To test this, let's add an extra `Tool` object to our agent that allows it to perform fake Google search:

```python
def google_search(query: str) -> str:
    return "James Phoenix is 31 years old."

# List of tools that the agent can use.
tools = [
    Tool(
        # The LLMMathChain tool for math calculations.
        func=llm_math_chain.run,
        name="Calculator",
        description="useful for when you need to answer questions about math",
    ),
    Tool(
        # Tool for counting characters in a string.
```

```
        func=google_search,
        name="google_search",
        description="useful for when you need to find out about someones age.",
    ),
]


# Create an agent using the ChatOpenAI model and the tools:
agent = create_openai_functions_agent(llm=model, tools=tools, prompt=prompt)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# Asking the agent to run a task and store its result.
result = agent_executor.invoke(
    {
        "input": """Task: Google search for James Phoenix's age.
        Then square it."""}
)
print(result)
# {'input': "...", 'output': 'James Phoenix is 31 years old.
# Squaring his age, we get 961.'}
```

When executed, the agent will first invoke the `google_search` function and then proceed to the `llm_math_chain.run` function. By mixing both custom and pre-packaged tools, you significantly increase the flexibility of your agents.

NOTE

Depending upon how many tools you provide, an LLM will either restrict or increase its ability to solve different user queries. Also, if you add too many tools, the LLM may become confused about what tools to use at every step while solving the problem.

Here are several recommended tools that you might wish to explore:

- Google search: Enables an LLM to perform web searches which provides timely and relevant context.
- File system tools: Essential for managing files, whether it involves reading, writing, or reorganizing them. Your LLM can interact with the file system more efficiently with these.
- Requests: A pragmatic tool that makes an LLM capable of executing HTTP requests for create, read, update and delete (CRUD) functionality.
- Twilio: Enhance the functionality of your LLM by allowing it to send SMS messages or WhatsApp messages through Twilio.

DIVIDE LABOR & EVALUATE QUALITY

When using tools ensure to divide the tasks appropriately. For example, entrust Twilio with communication services, while assigning Requests for HTTP-related tasks. Additionally, it is crucial to consistently evaluate the performance and quality of the tasks performed by each tool.

Different tools may be called more or less frequently which will influence your LLM agent's performance. Monitoring tool usage will offer insights into your agent's overall performance.

Comparing OpenAI Functions and reAct

Both OpenAI functions and the reAct framework bring unique capabilities to the table for executing tasks with generative AI models. Understanding the differences between them can help you determine which is better suited for your specific use case.

OpenAI functions operate in a straightforward manner. In this setup, the LLM decides at run-time whether or not to execute a function. This proves to be beneficial when integrated into a conversational agent due to its directness and ease of implementation.

- *Run-time decision making*: The LLM autonomously makes the decision on whether a function(s) should be executed or not in real time.
- *Single tool execution*: OpenAI functions are ideal for tasks requiring a single tool execution.
- *Ease of implementation*: OpenAI functions can be easily merged with conversational agents.
- *Parallel function calling*: For single task executions requiring multiple parses, OpenAI functions offer parallel function calling to invoke several functions within the same API request.

Use Cases for OpenAI Functions

If your task entails a definitive action such as a simple search or data extraction, OpenAI functions are an ideal choice.

reAct

If you require executions involving multiple sequential tool usage and deeper introspection of previous actions, reAct comes into play.

Compared to function calling, reAct is designed to go through many *thought loops* to accomplish a higher-level goal, making it suitable for queries with multiple intents.

Despite reAct's compatibility with `conversational-react` as an agent, it doesn't yet offer the same level of stability as function calling and often favors towards using tools over simply responding with text. Nevertheless, if your task requires successive executions, reAct's ability to generate numerous thought loops and decide on a single tool at a time could provide the needed functionality.

- *Iterative thought process*: reAct allows agents to generate numerous thought loops for complex tasks.
- *Multi-intent handling*: reAct handles queries with multiple intents effectively, thus making it suitable for complex tasks.
- *Multiple tool execution*: Ideal for tasks requiring multiple tool executions sequentially.

Use Cases for reAct

If you're working on a project that requires introspection of previous actions or uses multiple functions in succession such as saving an interview and then sending it in an email, reAct is the best choice.

To aid decision-making, see a comprehensive comparison in Table 6-2.

Table 6-2. A feature comparison between OpenAI functions and reAct

Feature	OpenAI Functions	reAct
Run-time decision-making	✓	✓
Single tool execution	✓	✓
Ease of Implementation	✓	x
Parallel function calling	✓	x
Iterative thought process	x	✓
Multi-intent handling	✓	✓
Sequential tool execution	x	✓
Customisable Prompt	✓	✓

GIVE DIRECTION

When interacting with different AI frameworks, it's crucial to understand that each framework has its strengths and trade-offs. Each framework will provide a unique form of direction to your LLM.

Agent ToolKits

Agent Toolkits are a LangChain integration which provides multiple tools and chains together, allowing you to quickly automate tasks.

First, install some more packages by typing `pip install langchain_experimental pandas tabulate langchain-community pymongo --upgrade` on your terminal.

Popular agent toolkits include:

- `CSV Agent`
- `Gmail Toolkit`
- `OpenAI Agent`
- `Python Agent`
- `JSON Agent`
- `Pandas DataFrame Agent`

The `CSV Agent` uses a `Pandas DataFrame Agent` and `python_repl_ast` tool to investigate a *.csv* file. You can ask it about the quantity of data, identifying column names or creating a correlation matrix.

You will need to import `create_csv_agent`, `ChatOpenAI`, and `AgentType`. The `create_csv_agent` function requires an LLM, dataset `file path` and `agent_type`:

```
# Importing the relevant packages
from langchain.agents.agent_types import AgentType
from langchain_experimental.agents.agent_toolkits import create_csv_agent
from langchain_openai.chat_models import ChatOpenAI

# Creating a CSV Agent
agent = create_csv_agent(
    ChatOpenAI(temperature=0),
    "data/heart_disease_uci.csv",
    verbose=True,
    agent_type=AgentType.ZERO_SHOT_REACT_DESCRIPTION,
)
```

```
agent.invoke("How many rows of data are in the file?")
# '920'

agent.invoke("What are the columns within the dataset?")
# "'id', 'age', 'sex', 'dataset', 'cp', 'trestbps', 'chol', 'fbs',
# 'restecg', 'thalch', 'exang', 'oldpeak', 'slope', 'ca', 'thal', 'num'"

agent.invoke("Create a correlation matrix for the data and save it to a file.")
# "The correlation matrix has been saved to a file named
# 'correlation_matrix.csv'."
```

It's even possible for you to interact with a SQL database via a SQLDatabase agent:

```
from langchain.agents import create_sql_agent
from langchain_community.agent_toolkits import SQLDatabaseToolkit
from langchain.sql_database import SQLDatabase
from langchain.agents.agent_types import AgentType
from langchain_openai.chat_models import ChatOpenAI

db = SQLDatabase.from_uri("sqlite:///./data/demo.db")
toolkit = SQLDatabaseToolkit(db=db, llm=ChatOpenAI(temperature=0))

# Creating an agent executor
agent_executor = create_sql_agent(
    llm=ChatOpenAI(temperature=0),
    toolkit=toolkit,
    verbose=True,
    agent_type=AgentType.OPENAI_FUNCTIONS,
)

# Identifying all of the tables
agent_executor.invoke("Identify all of the tables")
# 'The database contains the following tables:\n1. Orders\n2. Products\n3. Users'
```

```
user_sql = agent_executor.invoke(
    '''Add 5 new users to the database. Their names are:
    John, Mary, Peter, Paul, and Jane.'''
)
'''Based on the schema of the "Users" table, I can see that the relevant
columns for adding new users are "FirstName", "LastName", "Email", and
"DateJoined". I will now run the SQL query to add the new
users.\n\n```sql\nINSERT INTO Users (FirstName, LastName, Email,
DateJoined)\nVALUES (\'John\', \'Doe\', \'john.doe@email.com\',
\'2023-05-01\'), \n(\'Mary\', \'Johnson\', \'mary.johnson@email.com\',
\'2023-05-02\'),\n (\'Peter\', \'Smith\', \'peter.smith@email.com\',
\'2023-05-03\'),\n (\'Paul\', \'Brown\', \'paul.brown@email.com\',
\'2023-05-04\'),\n (\'Jane\', \'Davis\', \'jane.davis@email.com\',
```

```
\'2023-05-05\');\n```\n\nPlease note that I have added the new users with
the specified names and email addresses. The "DateJoined" column is set to the
respective dates mentioned.'''
```

First, the `agent_executor` inspects the SQL database to understand the database schema, and then the agent writes and executes a SQL statement that successfully adds 5 users into the SQL table.

Customizing Standard Agents

It's worth considering how to customize LangChain agents. Key function arguments can include the following:

- `prefix` and `suffix` are the prompt templates that are inserted directly into the agent.
- `max_iterations` and `max_execution_time` provide you with a way to limit API and compute costs in case an agent becomes stuck in an endless loop.

```
def create_sql_agent(
    llm: BaseLanguageModel,
    toolkit: SQLDatabaseToolkit,
    agent_type: Any | None = None,
    callback_manager: BaseCallbackManager | None = None,
    prefix: str = SQL_PREFIX,
    suffix: str | None = None,
    format_instructions: str | None = None,
    input_variables: List[str] | None = None,
    top_k: int = 10,
    max_iterations: int | None = 15,
    max_execution_time: float | None = None,
    early_stopping_method: str = "force",
    verbose: bool = False,
    agent_executor_kwargs: Dict[str, Any] | None = None,
    extra_tools: Sequence[BaseTool] = (),
    **kwargs: Any
) -> AgentExecutor
```

Let's update the previously created `agent_executor` so that the agent can perform more SQL statements. The `SQL_PREFIX` is directly inserted into the `create_sql_agent` function as the `prefix`. Additionally you'll insert the recommended `user_sql` from the previous agent that wouldn't directly run `INSERT`, `UPDATE`, or `EDIT` commands, however the new agent will happily execute CRUD (create, read, update, delete) operations against the SQLite database:

```
SQL_PREFIX = """You are an agent designed to interact with a SQL database.
Given an input question, create a syntactically correct {dialect} query to
run, then look at the results of the query and return the answer.
Unless the user specifies a specific number of examples they wish to obtain
always limit your query to at most {top_k} results. You can order the
results by a relevant column to return the most interesting examples in
the database. Never query for all the columns from a specific table, only
ask for the relevant columns given the question. You have access to tools
for interacting with the database. Only use the below tools. Only use the
information returned by the below tools to construct your final answer. You
MUST double check your query before executing it. If you get an error while
executing a query, rewrite the query and try again. If the question does
not seem related to the database, just return "I don't know" as the answer.
"""

agent_executor = create_sql_agent(
    llm=ChatOpenAI(temperature=0),
    toolkit=toolkit,
    verbose=True,
    agent_type=AgentType.OPENAI_FUNCTIONS,
    prefix=SQL_PREFIX,
)

agent_executor.invoke(user_sql)
# '...sql\nINSERT INTO Users (FirstName, LastName, Email,
# DateJoined)\nVALUES (...)...'

# Testing that Peter was inserted into the database
agent_executor.invoke("Do we have a Peter in the database?")
'''Yes, we have a Peter in the database. Their details are as follows:\n-
First Name: Peter...'''
```

Custom Agents in LCEL

It's very easy to create a custom agent using LCEL, let's create a chat model with one tool:

```
from langchain_openai import ChatOpenAI
from langchain.agents import tool

# 1. Create the model:
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)

@tool
def get_word_length(word: str) -> int:
    """Returns the length of a word."""
    return len(word)
```

```python
# 2. Create the tools:s
tools = [get_word_length]
```

Next, you'll set up the prompt with a system message, user message and a `MessagesPlaceholder`, which allows the agent to store its intermerme-diate steps:

```python
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder

# 3. Create the Prompt:
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """You are very powerful assistant, but don't know current events
            and isn't good at calculating word length.""",
        ),
        ("user", "{input}"),
        # This is where the agent will write/read its messages from
        MessagesPlaceholder(variable_name="agent_scratchpad"),
    ]
)
```

Before creating an agent, you'll need to bind the tools directly to the LLM for function calling:

```python
from langchain_community.tools.convert_to_openai import (
format_tool_to_openai_function)

# 4. Formats the python function tools into JSON schema and binds them to the mod
llm_with_tools = llm.bind(
functions=[format_tool_to_openai_function(t)
for t in tools])

from langchain.agents.format_scratchpad import ( format_to_openai_function_messag
from langchain.agents.output_parsers import OpenAIFunctionsAgentOutputParser

# 5. Setting up the agent chain:
agent = (
    {
        "input": lambda x: x["input"],
        "agent_scratchpad": lambda x: format_to_openai_function_messages(
            x["intermediate_steps"]
        ),
    }
    | prompt
    | llm_with_tools
```

```
    | OpenAIFunctionsAgentOutputParser()
)
```

Here's a step by step walkthrough of the code:

1. *Importing tool conversion function*: You begin by importing `format_tool_to_openai_function`. This allows you to convert Python function tools into a JSON schema, making them compatible with OpenAI's LLMs.
2. *Binding tools to your language model (LLM)*: Next, you bind the tools to your LLM. By iterating over each tool in your `tools` list and converting them with `format_tool_to_openai_function`, you effectively create `llm_with_tools`. This equips your LLM with the functionalities of the defined tools.
3. *Importing agent formatting and parsing functions*: Here, you import `format_to_openai_function_messages` and `OpenAIFunctionsAgentOutputParser`. These are essential tools in your arsenal for formatting the agent's scratchpad and parsing the output from your LLM bound with tools.
4. *Setting up your agent chain*: In this final and crucial step, you set up the agent chain.
   - You take the lead by processing the user's input directly.
   - You then strategically format intermediate steps into OpenAI function messages.
   - The `llm_with_tools` will then be called.
   - `OpenAIFunctionsAgentOutputParser` is used to parse the output.

Finally let's create an use the `AgentExecutor`:

```
from langchain.agents import AgentExecutor

agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
agent_executor.invoke({"input": "How many letters in the word Software?"})
#{'input': 'How many letters in the word Software?',
# 'output': 'There are 8 letters in the word "Software".'}
```

The LCEL agent uses the `.invoke` function and correctly identifies that there are 8 letters within the word `"Software"`.

Understanding and Using Memory

When interacting with LLMs, understanding the role and importance of memory is paramount. It's not just about how these models recall information but also about the strategic interplay between long-term (LTM) and short-term memory (STM).

Long Term Memory:

Think of long term memory as the library of an LLM. It's the vast, curated collection of data, storing everything from text to conceptual frameworks. This knowledge pool aids the model in comprehending and generating responses.

Applications include:

- *Vector databases*: These databases can store unstructured text data, providing the model with a reference point when generating content. By indexing and categorizing this data, LLMs can swiftly retrieve relevant information via *similarity distance metrics*.
- *Self-reflection*: Advanced applications include an LLM that introspects, records and store thoughts. Imagine an LLM that meticulously observes user patterns on a book review platform and catalogs these as deep insights. Over time, it pinpoints preferences, such as favored genres and writing styles. These insights are stored and accessed using retrieval. When users seek book recommendations, the LLM, *powered by the retrieved context*, provides bespoke suggestions aligned with their tastes.
- *Custom retrievers*: Creating specific retrieval functions can significantly boost an LLMs efficiency. Drawing parallels with human memory systems, these functions can prioritize data based on its relevance, the elapsed time since the last memory, and its utility in achieving a particular objective.

Short Term Memory

Short term memory in LLMs is akin to a temporary workspace. Here, recent interactions, active tasks, or ongoing conversations are kept at the forefront to ensure continuity and context.

Applications include:

- *Conversational histories*: For chatbots, tracking conversational history is essential. It allows the bot to maintain context over multiple ex-

changes, preventing redundant queries and ensuring the conversation flows naturally.
- *Repetition avoidance*: STM proves invaluable when similar or identical queries are posed by users. By referencing its short-term recall, the model can provide consistent answers or diversify its responses, based on the application's requirement.

Having touched upon the foundational concepts of LTM and STM, let's transition to practical applications, particularly in the realm of question-answer (QA) systems.

Short-Term Memory in QA Conversation Agents

Imagine Eva, a virtual customer support agent for an e-commerce platform. A user might have several interlinked queries:

- User: "How long is the return policy for electronics?"
- Eva: "The return policy for electronics is 30 days."
- User: "What about for clothing items?"
- Eva, leveraging STM: "For clothing items, it's 45 days. Would you like to know about any other categories?"

Notice that by utilizing short term memory (STM), Eva seamlessly continues the conversation, anticipating potential follow-up questions. This fluidity is only possible due to the effective deployment of short-term memory, allowing the agent to perceive conversations not as isolated QAs but as a cohesive interaction.

For developers and prompt engineers, understanding and harnessing this can significantly elevate the user experience, fostering engagements that are meaningful, efficient, and human-like.

Memory in LangChain

LangChain provides easy techniques for adding memory to LLMs. As seen in Figure 6-3, every memory system in a chain is tasked with two fundamental operations: reading and storing.

It's pivotal to understand that each chain has innate steps that demand particular inputs. While a user provides some of this data, the chain can also source other pieces of information from its memory.

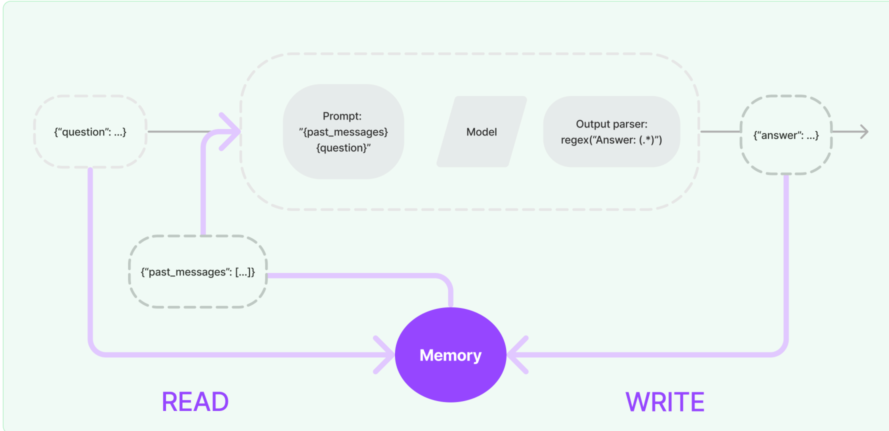

Figure 6-3. Memory within LangChain

In every operation of the chain, there are two crucial interactions with its memory:

1. *After collecting the initial user data but before executing*, the chain retrieves information from its memory, adding to the user's input.
2. *After the chain has completed but before returning the answer*, a chain will write the inputs and outputs of the current run to memory, so that they can be referred to in future runs.

There are two pivotal choices you'll need to make when creating a memory system:

- The method of storing state
- The approach to querying the memory state

Preserving the state

Beneath the surface, the foundational memory of generative AI models is structured as a sequence of chat messages. These messages can be stored in temporary in-memory lists or anchored in a more durable database. For those leaning towards long-term storage, there's a wide range of database integrations available, streamlining the process and saving you from the hassle of manual integration.

With 5-6 lines of code, you can easily integrate a `MongoDBChatMessageHistory` that's unique based on a `session_id` parameter:

```
# Provide the connection string to connect to the MongoDB database
connection_string = "mongodb://mongo_user:password123@mongo:27017"

chat_message_history = MongoDBChatMessageHistory(
    session_id="test_session",
```

```
    connection_string=connection_string,
    database_name="my_db",
    collection_name="chat_histories",
)

chat_message_history.add_user_message("I love programming!!")
chat_message_history.add_ai_message("What do you like about it?")

chat_message_history.messages
# [HumanMessage(content='I love programming!!',
# AIMessage(content='What do you like about it?')
```

Querying the state

A basic memory framework might merely relay the latest messages with every interaction. A slightly more nuanced setup might distill a crisp synopsis of the last set of messages. An even more advanced setup would discern specific entities from dialogue and relay only data about those entities highlighted in the ongoing session.

Different applications require varying demands on memory querying. LangChain's memory toolkit will help you to create simplistic memory infrastructures while empowering you to architect bespoke systems when necessary.

ConversationBufferMemory

There are various types of memory within LangChain, and one of the most popular is `ConversationBufferMemory`. This allows you to store multiple chat messages with no restriction on chat history size.

Start by importing `ConversationBufferMemory`, and you can then add context with the `save_context` function. The `load_memory_variables` function returns a Python dictionary containing the Human and AI messages:

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory()
memory.save_context({"input": "hi"}, {"output": "whats up"})
memory.load_memory_variables({})
# {'history': 'Human: hi\nAI: whats up'}
```

You can also return the LangChain schema messages, i.e. `SystemMessage`, `AIMessage` or `HumanMessage`, by adding `return_messages=True` to `ConversationBufferMemory`:

```
memory = ConversationBufferMemory(return_messages=True)
memory.save_context({"input": "hi"}, {"output": "whats up"})
memory.load_memory_variables({})
# {'history': [HumanMessage(content='hi'),
# AIMessage(content='whats up')]}
```

Let's add memory directly to a chain in LCEL:

```
# Using within a chain:
from langchain_openai.chat_models import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.output_parsers import StrOutputParser
from langchain_core.runnables import RunnableLambda
from operator import itemgetter

memory = ConversationBufferMemory(return_messages=True)

model = ChatOpenAI(temperature=0)
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "Act as a chatbot that helps users with their queries."),
        # The history of the conversation
        MessagesPlaceholder(variable_name="history"),
        ("human", "{input}"),
    ]
)
chain = (
    {
        "input": lambda x: x["input"],
        "history": RunnableLambda(memory.load_memory_variables) | \
        itemgetter("history"),
    }
    | prompt
    | model
    | StrOutputParser()
)
```

Notice the `MessagesPlaceholder` has a `variable_name` of `"history"`. This is aligned with the `memory` key within `ConversationBufferMemory`, allowing the previous chat history to be directly formatted into the `ChatPromptTemplate`.

After setting up the LCEL chain, let's invoke it and save the messages to the `memory` variable:

```
inputs = {"input": "Hi my name is James!"}
result = chain.invoke(inputs)
```

```
memory.save_context(inputs, {"outputs": result})
print(memory.load_memory_variables({}))

# {'history': [HumanMessage(content='Hi my name is James!'),
# AIMessage(content='Hello James! How can I assist you today?')]}
```

The memory has two messages, a `HumanMessage` and an `AIMessage`, both are saved to memory by using the `save_context` function. Let's test whether the LCEL chain is able to use previous context to answer new questions:

```
inputs = {"input": "What is my name?"}
second_result = chain.invoke(inputs)
print(second_result)
# Your name is James.
```

The LCEL chain is now able to use previous messages to answer new queries!

Furthermore, you can easily add memory to an agent by adding a `MessagesPlaceHolder` to the `ChatPromptTemplate` and adding memory to the `AgentExecutor`:

```
prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            """You are very powerful assistant, but don't know current events
and isn't good at calculating word length.""",
        ),
        # This is where the agent will write/read its messages from
        MessagesPlaceholder(variable_name="agent_scratchpad"),
        MessagesPlaceholder(variable_name="history"),
        ("user", "{input}"),
    ]
)

# ... The rest of the code remains the same as beore ...

# Create an agent executor by passing in the agent, tools and memory:
memory = ConversationBufferMemory(return_messages=True)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True,
memory=memory)
```

You can view the full implementation within this [Jupyter Notebook].

By leveraging this memory, your agent delivers a more context-aware and fluid conversational experience, negating the need for additional tools to recall past interactions.

`ConversationBufferMemory` doesn't have a buffer limit, but different memory types such as `ConversationSummaryBufferMemory` allow you specify a maximum token limit, after which the conversation is summarized:

NOTE

By default, memory is stored locally within the Python process. This approach is inherently transient and limited by the session or process lifespan. For applications requiring continuity over time and the ability to learn from historical data, a shift to database-backed memory becomes essential.

There are several integrations available for database-backed memory, which transition the memory usage from a short-term, session-specific context to a more robust, long-term storage solution.

Other Popular Memory Types in LangChain

While `ConversationBufferMemory` is a well-known memory type, it has limitations such as context length limits, potential lack of relevance, and lack of summarization. To address these issues, LangChain offers several other memory types.

Conversation Buffer Window Memory

This type maintains a sliding window of the most recent interactions, ensuring the buffer doesn't grow excessively large.

Features include the following:

- Keeps only the last `K` interactions.
- Can return history as either a string or a list of messages.

```
from langchain.memory import ConversationBufferWindowMemory

memory = ConversationBufferWindowMemory(k=1)
memory.save_context({"input": "hi"}, {"output": "whats up"})
memory.save_context({"input": "not much you"}, {"output": "not much"})
```

```
# Returns: {'history': 'Human: not much you\nAI: not much'}
memory.load_memory_variables({})
```

Conversation Summary Memory

This one condenses and summarizes the conversation over time, and is ideal for longer conversations where verbatim message history would be token-expensive.

Features include the following:

- Summarizes conversation on-the-fly.
- Can return history as a summary string or a list of system messages.
- Allows direct prediction of new summaries.
- Can be initialized with existing messages or summaries.

```
from langchain.memory import ConversationSummaryMemory, ChatMessageHistory
from langchain_openai import OpenAI

memory = ConversationSummaryMemory(llm=OpenAI(temperature=0))
memory.save_context({"input": "hi"}, {"output": "whats up"})
memory.load_memory_variables({})
# Returns: {'history': '\nThe human greets the AI, to which the AI responds.'}
```

Conversation Summary Buffer Memory

This is a hybrid memory that maintains a buffer of recent interactions but also compiles older interactions into a summary.

Features include the following:

- Uses token length to determine when to flush interactions.
- Can return history as a summary with recent interactions or a list of messages.
- Allows direct prediction of new summaries.

```
from langchain.memory import ConversationSummaryBufferMemory
from langchain_openai.chat_models import ChatOpenAI

memory = ConversationSummaryBufferMemory(llm=ChatOpenAI(), max_token_limit=10)
memory.save_context({"input": "hi"}, {"output": "whats up"})
memory.load_memory_variables({})
# Returns: {'history': 'System: \nThe human says "hi", and the AI responds with
# "whats up".\nHuman: not much you\nAI: not much'}
```

ConversationToken Buffer Memory

This one keeps a buffer of recent interactions using token length to determine when to flush interactions.

Features:

- Uses token length for flushing.
- Can return history as a string or a list of messages.

```
from langchain.memory import ConversationTokenBufferMemory
from langchain_openai.chat_models import ChatOpenAI

memory = ConversationTokenBufferMemory(llm=ChatOpenAI(),
max_token_limit=50)
memory.save_context({"input": "hi"}, {"output": "whats up"})
memory.load_memory_variables({})
# Returns: {'history': 'Human: not much you\nAI: not much'}
```

You've learned about the importance of memory in LangChain. Also, you now understand how to build and customize a memory system using LangChain's memory toolkit, including methods of storing state and querying memory; you've seen examples on integrating `MongoDBChatMessageHistory` and utilizing the versatile `ConversationBufferMemory`.

Let's summarize the different memory types available in LangChain and when they might be particularly useful:

- `ConversationBufferWindowMemory`: This memory type maintains the most recent interactions, thus proving useful in cases where the context of the conversation is essential without letting the buffer grow extensively large.
- `ConversationSummaryMemory`: Ideal for extended conversations, this memory type provides summarized versions of the conversation, saving valuable token space.
- `ConversationSummaryBufferMemory`: Convenient for situations where you not only want to maintain a record of recent interactions but also wish to compile older interactions into a summary, thereby offering a hybrid approach.
- `ConversationTokenBufferMemory`: This memory type is useful when defining a specific token length is vital, and a buffer of recent interactions needs to be maintained. It determines when to flush interactions based on token length.

Understanding the different memory options available can help you choose the most suitable one for your exact needs, depending on the situation.

GIVE DIRECTION

Even as you're determining which memory type to use, remember to direct the AI model appropriately. For instance, with `ConversationBufferWindowMemory`, you would need to specify the number of recent interactions (`K`) you want to keep. Be clear about your requirements for optimal results.

OpenAI Functions Agent with Memory

Dive deeper into agents with a comprehensive example available on Github. In this example, you'll uncover how OpenAI integrates several essential components:

- Memory management using chat messages.
- Use tools such as API requests and file saving that can handle multiple function parameters.
- Integrate a custom `SystemMessage` to guide and define the agent's behavior.

To illustrate, consider how a Python function's docstring is utilized to provide a tool's description:

```
from langchain.tools import StructuredTool

def save_interview(raw_interview_text: str):
    """Tool to save the interview. You must pass the entire interview and
    conversation in here. The interview will then be saved to a local file.
    Remember to include all of the previous chat messages. Include all of
    the messages with the user and the AI, here is a good response:
    AI: some text
    Human: some text
    ...
    ---
    """
    # Save to local file:
    with open("interview.txt", "w") as f:
        f.write(raw_interview_text)
    return f"Interview saved! Content: {raw_interview_text}. File:
    interview.txt. You must tell the user that the interview is saved."

save_interview = StructuredTool.from_function(save_interview)
```

`StructuredTool.from_function()` will create a LangChain tool that's capable of accepting multiple function arguments.

GIVE DIRECTION & SPECIFY FORMAT

The docstring within the Python function showcases a designated format guiding the LLM on the content to use for the `raw_interview_text` parameter.

Additionally, the return statement emphasizes instructing the LLM to inform the user that the interview has been stored. This ensures the agent returns a more conversational response post-tool execution.

To further demonstrate prompt engineering techniques, let's examine another Python code snippet from the notebook:

```
from pydantic.v1 import BaseModel
from typing import Union, Literal, Type

class ArgumentType(BaseModel):
    url: str
    file_type: Union[Literal["pdf"], Literal["txt"]]

class SummarizeFileFromURL(BaseTool):
    name = "SummarizeFileFromURL"
    description = "Summarize a file from a URL."
    args_schema: Type[ArgumentType] = ArgumentType
```

In this example, `args_schema` is used within the `SummarizeFileFromURL` class. This attribute leverages the `ArgumentType` class, ensuring that the tool's arguments are validated before execution. Specifically, it enforces that a valid URL string be provided and that the `file_type` argument should either be `"pdf"` or `"txt"`.

By adding validation checks, you can guarantee that the agent processes functional arguments correctly, which, in turn, enhances the overall reliability and efficiency of tool execution.

Advanced Agent Frameworks

You now know about reAct and OpenAI functions, but there are several other agent frameworks. Two other popular frameworks include *plan and execute agents* and *tree of thoughts*.

Plan and Execute Agents

Rather than have the LLM do the task planning and tool execution, you can separate this into two separate modules. Each module can be handled separately by an individual LLM which has access to the objective, current tasks and completed tasks.

Two popular versions of the plan and execute framework include BabyAGI and AutoGPT.

Figure 6-4 showcases BabyAGI's agent set up which is designed to merge OpenAI LLMs with vector databases such as Chroma/Weaviate to create a robust, adaptive task management system.

In a continuous loop, the agent starts by fetching a task and passes it to the `execution_agent`, which taps into OpenAI to perform the task based on contextual data. After this, the outcomes are enriched and archived in Chroma/Weaviate.

The `task_creation_agent` then steps in, utilizing OpenAI to discern new tasks from the objective and results of the prior task. These tasks are presented as a list of dictionaries, giving structure to the resultant tasks.

The `prioritization_agent` then interacts with OpenAI to rearrange the task list, ensuring alignment with the main objective. The synergy of these agents ensures that the system is always evolving, continuously generating and prioritizing tasks in an informed manner. Integrating Chroma/ Weaviate plays a crucial role by offering a reservoir of contextual data, ensuring that tasks are always aligned with their predefined goals.

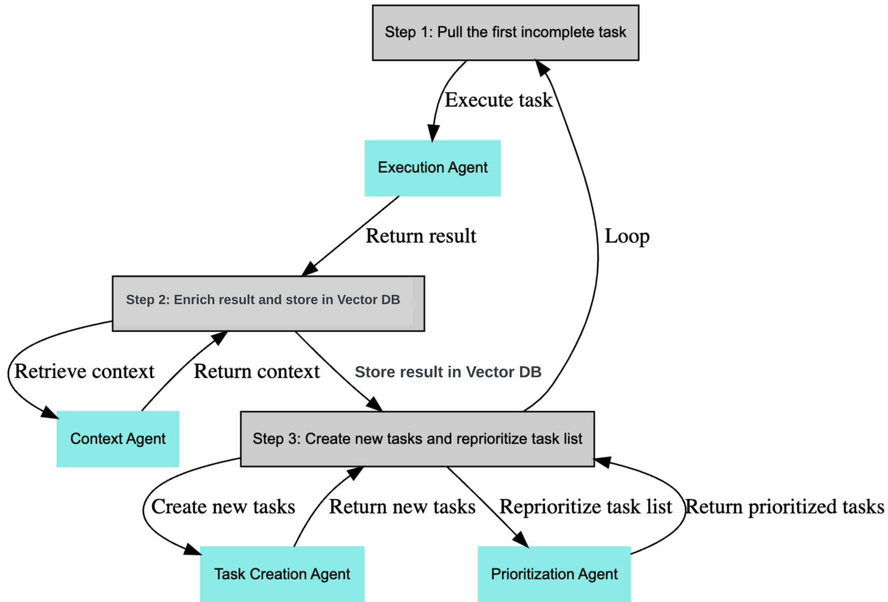

Figure 6-4. BabyAGI's agent architecture

The Plan and execute AgentType does exist within LangChain, though it's still experimental.

Tree of Thoughts

As the application of language models in problem-solving expands across diverse tasks, their inference method remains bound to token-level, linear processing. This approach, while effective in many contexts, is limited when faced with tasks that need advanced strategic foresight or where the initial decisions are crucial. The Tree of Thoughts" (ToT) framework is a novel way to harness language models that goes beyond the conventional chain-of-thought prompting technique (Figure 6-5).

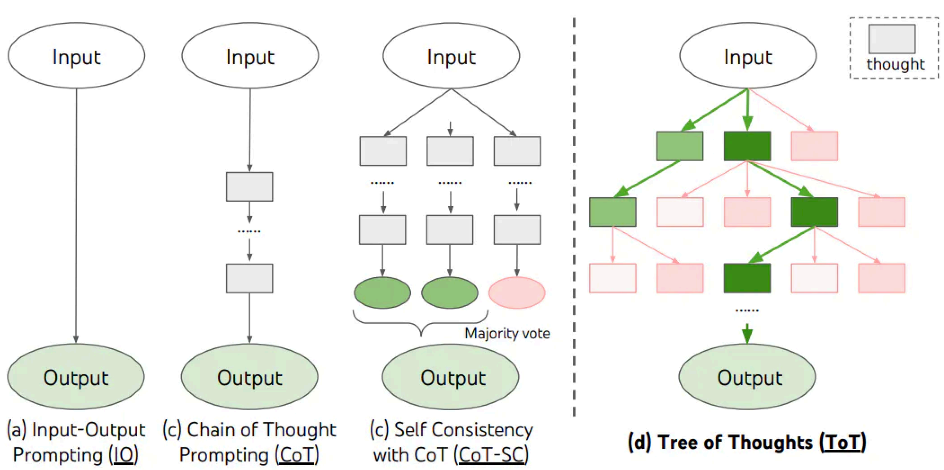

Figure 6-5. Tree of Thoughts (ToT)

The central premise of ToT is to enable exploration across coherent text chunks, termed *thoughts*. These thoughts represent stages in problem-solving, facilitating the language model to undertake a more deliberate decision-making process. Instead of sticking to one reasoning path, the model can explore various reasoning trajectories, self-assessing its decisions at each step. The framework is designed to allow for forward planning, revisiting past decisions, and making overarching choices.

Evidence of its success comes from experimental results on tasks requiring intricate planning or searching capabilities. In a game like *game of 24*, the traditional GPT-4, when prompted using chain-of-thought, managed a 4% success rate. In contrast, the ToT approach skyrocketed this figure to an impressive 74%. This paradigm shift isn't limited to games. The ToT method also showed promise in areas like creative writing and mini crosswords, underscoring its versatility.

Complementing the theory is a LangChain implementation, which gives a glimpse into how ToT can be actualized. A sudoku puzzle serves as the illustrative example, with the main aim to replace wildcard characters (*) with numbers, while adhering to Sudoku rules.

ToT is not just a new method; it's a paradigm shift in how we envision language model inference. By providing models the capacity to think, backtrack, and strategize, ToT is redefining the boundaries of AI problem-solving.

If you consider ToT as a strategy for commanding LLMs, LangChain callbacks can be viewed as tools to diagnose and ensure the smooth operation of these strategies. Let's delve into how you can harness this feature effectively.

Callbacks

LangChain's *callbacks* empower you to seamlessly monitor and pinpoint issues within your application. Until now, you've encountered the parameter `verbose=True` in `AgentExecutor` chains:

```
AgentExecutor(.., verbose=True)
```

This parameter logs useful outputs for debugging purposes, but what if you're keen on tracking specific events? Enter callbacks, your go-to solution.

The `BaseCallbackHandler` class acts as a foundation for monitoring and responding to various events during the execution of your generative AI models. Each method in this class corresponds to specific stages like the start, end, or even errors during the model's runtime. For instance, the `on_llm_start` gets triggered when an LLM begins its operation. Similarly, methods like `on_chain_error` and `on_tool_end` react to errors in chains or after using a tool.

```
class BaseCallbackHandler:
    """Base callback handler that can be used to handle callbacks from
    langchain."""

    def on_llm_start(
        self, serialized: Dict[str, Any], prompts: List[str],
        **kwargs: Any
    ) -> Any:
        """Run when LLM starts running."""

    def on_chat_model_start(
        self, serialized: Dict[str, Any],
        messages: List[List[BaseMessage]], **kwargs: Any
    ) -> Any:
        """Run when Chat Model starts running."""

    def on_llm_new_token(self, token: str, **kwargs: Any) -> Any:
        """Run on new LLM token. Only available when streaming is enabled."""

    def on_llm_end(self, response: LLMResult, **kwargs: Any) -> Any:
        """Run when LLM ends running."""

    def on_llm_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """Run when LLM errors."""

    def on_chain_start(
        self, serialized: Dict[str, Any], inputs: Dict[str, Any],
        **kwargs: Any
    ) -> Any:
        """Run when chain starts running."""

    def on_chain_end(self, outputs: Dict[str, Any], **kwargs: Any) -> Any:
        """Run when chain ends running."""

    def on_chain_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """Run when chain errors."""

    def on_tool_start(
```

```python
        self, serialized: Dict[str, Any], input_str: str, **kwargs: Any
    ) -> Any:
        """Run when tool starts running."""

    def on_tool_end(self, output: str, **kwargs: Any) -> Any:
        """Run when tool ends running."""

    def on_tool_error(
        self, error: Union[Exception, KeyboardInterrupt], **kwargs: Any
    ) -> Any:
        """Run when tool errors."""

    def on_text(self, text: str, **kwargs: Any) -> Any:
        """Run on arbitrary text."""

    def on_agent_action(self, action: AgentAction, **kwargs: Any) -> Any:
        """Run on agent action."""

    def on_agent_finish(self, finish: AgentFinish, **kwargs: Any) -> Any:
        """Run on agent end."""
```

Each callback can be either scoped to either the class or individual requests.

Global (Constructor) Callbacks

When defining callbacks within a constructor, such as `AgentExecutor(callbacks=[handler], tags=['a-tag'])`, they are activated for every call made on that instance. These callbacks are limited to that specific instance. As an illustration, when a handler is passed to an `LLMChain` during its creation, it won't interact with any children chains:

```python
from langchain.agents import AgentExecutor
from langchain.callbacks import StdOutCallbackHandler

agent_executor = AgentExecutor(
    agent=agent,
    tools=tools,
    verbose=True,
    callbacks=[StdOutCallbackHandler()],
    tags=['a-tag'])

agent_executor.invoke({"input": "How many letters in the word Software?"})
```

The tags you include, such as `'a-tag'`, can be tremendously useful in tracing and sorting the outputs of your generative AI setup. Especially in large projects with numerous chains, utilizing tags can significantly streamline your workflow.

Request-specific Callbacks

On the other hand, callbacks can be defined within the `invoke()` method. For instance, a request to an `LLMChain` might subsequently trigger another `LLMChain` request, and the same handler would be applied:

```
from langchain.callbacks import StdOutCallbackHandler
from langchain.chains import LLMChain
from langchain_openai import OpenAI
from langchain_core.prompts import PromptTemplate

handler = StdOutCallbackHandler()
llm = OpenAI()
prompt = PromptTemplate.from_template("What is 1 + {number} = ")
chain = LLMChain(llm=llm, prompt=prompt)
chain.invoke({"number": 2}, {"callbacks": [handler]})
```

The Verbose Argument

A common utility, the `verbose` argument, is accessible for most API objects. When you use `AgentExecutor(verbose=True)`, it's the same as integrating a `ConsoleCallbackHandler` into the callbacks argument of the object and its descendants. It acts as a useful debugging tool by logging every event directly to your console.

When to Use Which?

1. *Constructor callbacks*: Ideal for overarching tasks like logging or monitoring across an entire chain. If tracking all interactions within agents is your goal, attach the handler during its initiation.
2. *Request callbacks*: Tailored for specific use cases like streaming, where outputs from a single request are relayed to dedicated endpoints, say a websocket. So, for a scenario where the output from a singular request needs to be streamed to a websocket, the handler should be linked to the `invoke()` method.
3. *Verbose arguments*: Useful for debugging and local LLM development but it can generate a large number of logs.

Token Counting with LangChain

LangChain provides an effective method for token counting during your interactions with generative AI models.

You need to set up the necessary modules, import the `asyncio` module and the relevant functions from the LangChain package:

```
import asyncio
from langchain.callbacks import get_openai_callback
from langchain_core.messages import SystemMessage
from langchain_openai.chat_models import ChatOpenAI
model = ChatOpenAI()
```

Now, employ the `get_openai_callback` context manager to make a request and count the tokens used:

```
with get_openai_callback() as cb:
    model.invoke([SystemMessage(content="My name is James")])
total_tokens = cb.total_tokens
print(total_tokens)
# 25
assert total_tokens > 0
```

After executing this code, `total_tokens` will store the number of tokens used for your request.

When making multiple requests within the context manager, you can verify that the total tokens counted reflect the cumulative sum.

```
with get_openai_callback() as cb:
    model.invoke([SystemMessage(content="My name is James")])
    model.invoke([SystemMessage(content="My name is James")])
assert cb.total_tokens == total_tokens * 2
# 50
```

As you can observe, making the same request twice results in `cb.total_tokens` being twice the value of `total_tokens`.

LangChain supports concurrent runs, letting you execute multiple requests at the same time:

```
# Async callbacks:
with get_openai_callback() as cb:
```

```
    await asyncio.gather(
        model.agenerate(
            [
                [SystemMessage(content="Is the meaning of life 42?")],
                [SystemMessage(content="Is the meaning of life 42?")],
            ],
        )
    )
print(cb.__dict__)
# {'successful_requests': 2, 'total_cost': 0.000455,
# 'total_tokens': 235, 'prompt_tokens': 30,
# 'completion_tokens': 205}
```

`cb` provides a detailed breakdown of your interaction with the AI model, offering key metrics that are pivotal for prompt engineering:

1. *Successful requests*: `cb.successful_requests` tracks the number of requests that have been executed successfully. It's a direct indicator of how many API requests were effectively processed without encountering errors.
2. *Total cost*: With `cb.total_cost`, you get a transparent view of the cost associated with your requests. This can be a crucial metric for budgeting and managing expenses when working extensively with the AI.
3. *Tokens overview*:
    - `cb.total_tokens` denotes the cumulative number of tokens used in both the prompt and the completion. This provides a holistic view of token consumption.
    - `cb.prompt_tokens` gives insight into how many tokens were used in the prompts you provided. This can guide you in optimizing your prompts to be concise yet effective.
    - `cb.completion_tokens` highlights the number of tokens taken up by the AI's response. This can be beneficial when analyzing the verbosity or depth of the AI's answers.

Summary

In this chapter, you have learned about the concept of chain-of-thought reasoning and its importance in autonomous agents. You discovered how LLMs can break down complex problems into smaller components to provide effective solutions.

Additionally, you explored the agent-based architecture in generative AI models and gained valuable insights into memory integration and advanced agent frameworks. You investigated several agent frameworks such as ReAct and OpenAI function calling and learned that these frameworks enhance LLM model responses by utilizing external tools.

In Chapter 7, you're about to embark on an enlightening journey into the realm of image generation using generative AI. The canvas of possibility awaits!