**3. Chains, Tools and Agents**

Aarushi Kansal[**1**](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_3_Chapter.xhtml#Aff2)

(1)

Melbourne, Australia

In Chapter [2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml), you learned about RAG, memory, retrieval, and embeddings. You were able to combine these concepts and build yourself a command-line chatbot that answered your questions *and* could remember the rest of your conversation. This allowed the LLM to become “smarter” by getting context from history. Your chatbot also had access to up-to-date, personal information via a vector database, meaning it was able to answer questions beyond what it was trained on. This also helped prevent hallucination.

Now you’re going to take it one step further and build an agent – an independent application that can access the world and make its own decisions on what steps to take to get to the final goal.

**High-Level Concepts**

Before going straight to code, I want to walk you through some theory on the concepts you’ll be making use of. In particular, I’ll talk you through chains, tools, and agents.

**Chains**

First concept (which you actually used briefly in Chapter [2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml)) is a chain. These are wrappers around multiple various components – ranging from LLMs, APIs, libraries, databases, utility functions, etc. They are one of the core components of LangChain and enable you to really augment your LLM in a structured and easy way. You can craft your own chains or you can use the many existing ones. Chains are super important because they allow you to become a lot more creative with LLMs and solve increasingly complex problems, through integrating various entities.

These chains can be really simple such as just having one LLM or they can be increasingly complex – by combining multiple entities (also sometimes called utility chains).

The ones you’ve already used are related to RAG and conversational history. Recall in Chapter [2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml) you used the following:

* ConversationChain
* ConversationalRetrievalChain

The conversation chain you used, extended another, simpler chain, called an LLMChain, which alone, just receives a prompt and LLM and makes the call to the specified LLM and spits out the output. This is one of the most basic chains, and the ConversationChain builds on this algorithm to load historical context into the prompt that is then passed into the LLMChain and queried.

The next one you used was ConversationalRetrievalChain, and it is a chain specifically for retrieving information from a data source (in Chapter [2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml), that data source was Weaviate). This one is a little more complex, as it does three major things:

* It takes the chat history in and crafts an entirely new question based on history and new query.
* This question is passed into the retriever (i.e., this becomes the query to Weaviate).
* After getting the right documents, it passes the original question and fetched documents into the LLM to get a response.

These are just two chains; there are a lot more available for you to use – I recommend you check them out and build some on top of them for even more customized use cases.

Okay, so now you understand the concept of chains and glueing multiple utilities together.

Think of chains the same as a human body lifting an arm, or yawning, or lifting a mug and drinking from it – all one smooth action, but a lot of little things happening and interacting with each other under the hood to make them happen.

And if chains are larger actions, you can think of tools as something that enhances your abilities and/or knowledge, for example, the ability to do complex math or execute Python code.

So now let’s talk about the tools that give your LLM further access to the world.

**Tools**

Tools are wrappers that allow your LLM to interact with the world. This is a fancy way of saying; these are essentially functions that take some sort of input and output something based on it.

For example, if you were using a search tool – your input might be a query like “best sushi restaurants in London,” and the output you would get is a list of top sushi restaurants in London. This is information that could then be fed into your LLM and used further – maybe to transform that list into a “tour of London’s best sushi places” or maybe make recommendations to your user based on their dietary needs.

These tools can be simple API calls, chains themselves, agents, or anything else that *does* something when given an input.

And again, LangChain provides you with a range of pre-built tools, for example:

* Search tools
* Bash script tools
* YouTube tools
* Weather tools
* Python REPL

And many more – check out their documentation for the full list.

For the majority of use cases, a combination of these pre-built tools should get you where you need, but you can also craft your own tools if needed.

**Building a Custom Tool**

Building your own tool is fairly simple, and you can get as complex with it as you need.

Within the LangChain framework, you have the BaseTool class, which is your blueprint to building a tool.

The main components of this blueprint are as follows:

* Name
* Description
* \_run function – Default function that runs when tool is called
* \_arun function – Function if you want async running

The name and description are required fields, and there are a few more optional fields that you can check out in the library if interested.

Let’s discuss the description field though – this is one of the most important fields because it’s what your LLM uses to make the decision on when/how and why to use this tool.

Some best practices for you to consider:

* Clearly state when to use the tool.
* State how (especially if it’s a more complicated tool).
* State when to *not* use the tool.
  + I’ve found this one very useful when using multiple tools inside of an agent. By being clear on when to not use the tool, you can really assist your LLM in becoming more accurate, as LLMs have the tendency to also just use a tool if they’re not sure exactly which one to use or if there isn’t one that best matches its need.
    - Provide some examples of using the tool.
  + This one is great for helping your LLM reason by seeing.

Okay, let’s see some code. This one is going to be a really simple tool that just reverses any string passed into it.

As you can see in the following, you have to extend the BaseTool class, provide a name and description, and implement the \_run method. I have not implemented the async function – but you definitely can for your own use case, if needed.

from langchain.tools import BaseTool

class StringReverseTool(BaseTool):

   name = "String Reversal Tool"

   description = "use this tool when you need to reverse a string"

def \_run(self, word: str):

   return word[::-1]

def \_arun(self, word: str):

   raise NotImplementedError("Async not supported by StringReverseTool")

Now that you understand chains and tools, I want to show you agents, which is where LLM development gets really interesting. Everything you’ve learned so far can be combined into one or multiple agents.

**Agents**

Agents are one of the most interesting, creative, and kind of buzzy concepts in the AI space currently. They are super powering and completely transforming the way we do complex tasks – previously only considered doable by human beings.

In more concrete terms, an agent is an application that is powered by an LLM and interacts with different APIs, entities, libraries, chains, tools, etc.

The LLM is the “brain” that makes the decisions on which chain and/or tools to execute, what to do with the output, and how to interpret various inputs/outputs and human interactions.

When ChatGPT first came out (as well as the other chatbots out there), the main way of interacting with GPT-3 was going on to OpenAI, writing some prompt, getting an answer, and then maybe continuing questioning or asking for different formatting, more in-depth answers, clarification, etc. This is a very manual and human process. In this process, the human is the decision maker, and the human has a goal or task to achieve.

Agents, on the other hand, attempt to replicate this human goal-oriented behavior – given a goal, they will determine their own tasks to achieve the tools to use and how to process the output of the tools and craft their own prompts to help them achieve each step to their final goal.

So what is this independent, self-thinking application actually made of?

1. 1.

An LLM

1. 2.

Memory

1. 3.

An agent

1. 4.

One or more tools

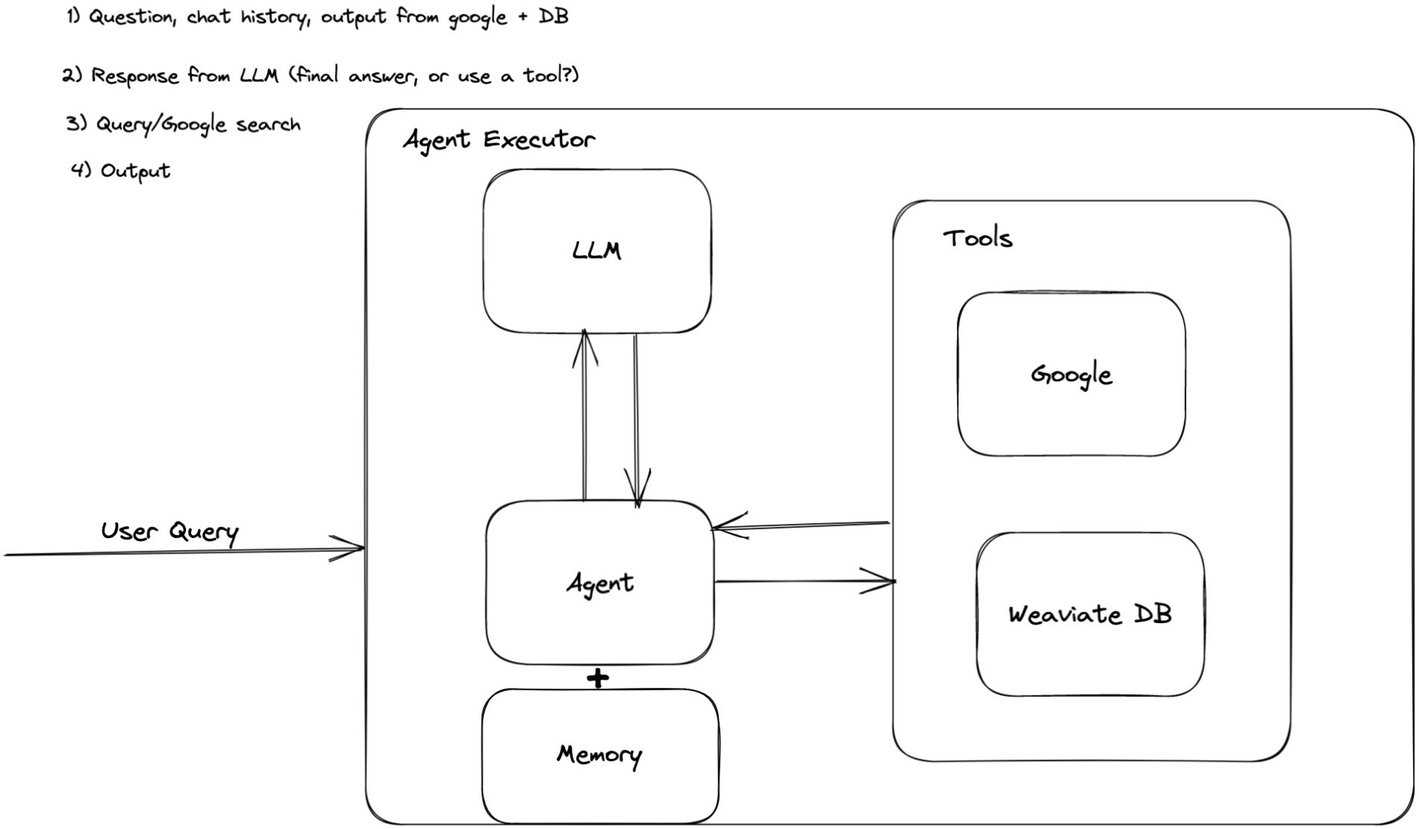
1. 5.

Agent executor

* 1. a.

This is what runs the actual code when told to do so.

You can see the architecture in Figure [3-1](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_3_Chapter.xhtml#Fig1)



***Figure 3-1***

Architecture for an agent – an independent, self-thinking application

Let’s dig a little into this component called an agent. At the crux of it, an agent is really a way of forcing the LLM to “think,” that is, a way of prompting the LLM to think in a certain style. For example, a very simple way would be to just say “think step by step” after asking a question. And since the onset of AI summer, there have been numerous papers on various algorithms and styles of prompting LLMs to facilitate better logic and reasoning and minimize hallucination.

LangChain provides a range of these pre-built for you. Let’s dive into some of the most commonly used ones:

* zero-shot-react-description
* react-docstore
* conversational-react-description
* chat-zero-shot-react-description
* chat-conversational-react-description
* self-ask-with-search

Here, maybe you’ve noticed a lot of them seem to have the word “react” in them. This is a fairly new framework for prompting LLMs. So let’s talk about the basic premises of ReAct.

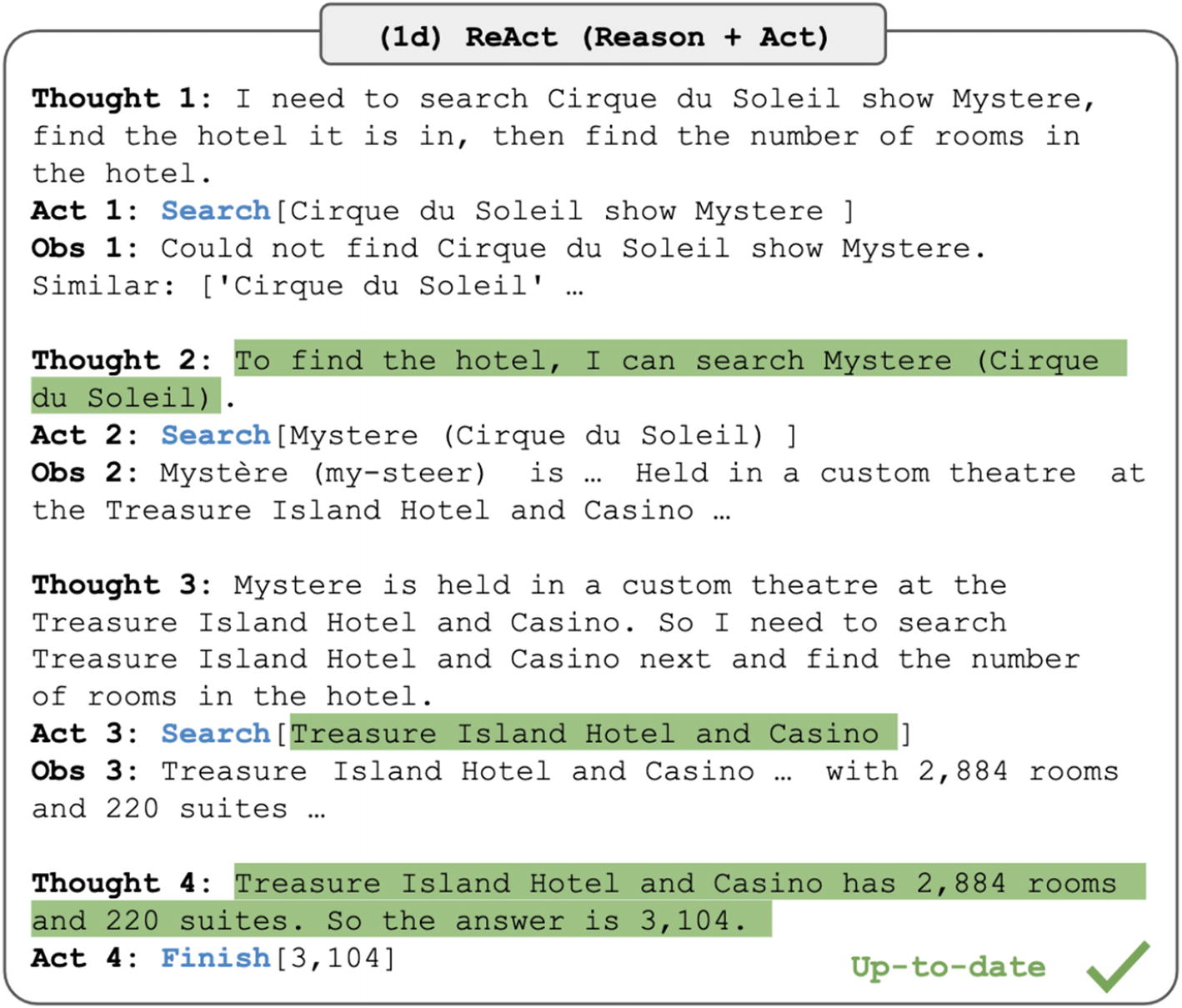
**ReAct**

ReAct stands for reason and act, and it’s a framework that was proposed in March 2023 and has gained significant traction since then.

You can read the full paper here: <https://arxiv.org/pdf/2210.03629.pdf>.

The goal of ReAct is to create a train of reasoning along with actions based on that reasoning and interweaving the two – meaning reasoning something, based on that taking an action and then based on the actions output reasoning again and taking another action until the task is achieved.

This is shown in Figure [3-2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_3_Chapter.xhtml#Fig2). Here you can see the LLM reasons that it needs to search Cirque du Soleil, find the hotel, and then find the number of rooms in the hotel. It then takes different actions and “observes” the output and based on that reasons or thinks to itself again and makes another action until it finally comes up with an answer.



***Figure 3-2***

Reason + Act examples from the original ReAct paper

And this concept is what most of the main agents are based on. The primary difference is what they are optimized for normally through tools, memory, and vector databases.

**zero-shot-react-description**

This agent type has no memory; it can only execute on one interaction. It will reason and output based on the ReAct framework but will not remember its previous thinking, or final answer on any subsequent uses.

**conversational-react-description**

This agent is the next enhancement on the zero shot agent – it allows for a memory. You can plug in any kind of memory. Recall in Chapter [2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml), you used ConversationBufferMemory and ConversationSummaryMemory; you could use these or you could also use an external storage as memory.

**react-docstore**

This agent uses ReAct but is optimized to use something called a Docstore in the context of LangChain. Basically an agent that uses some document store as a tool and can search in it for more context. The built-in ones include Wikipedia and In Memory (a Python dict representation).

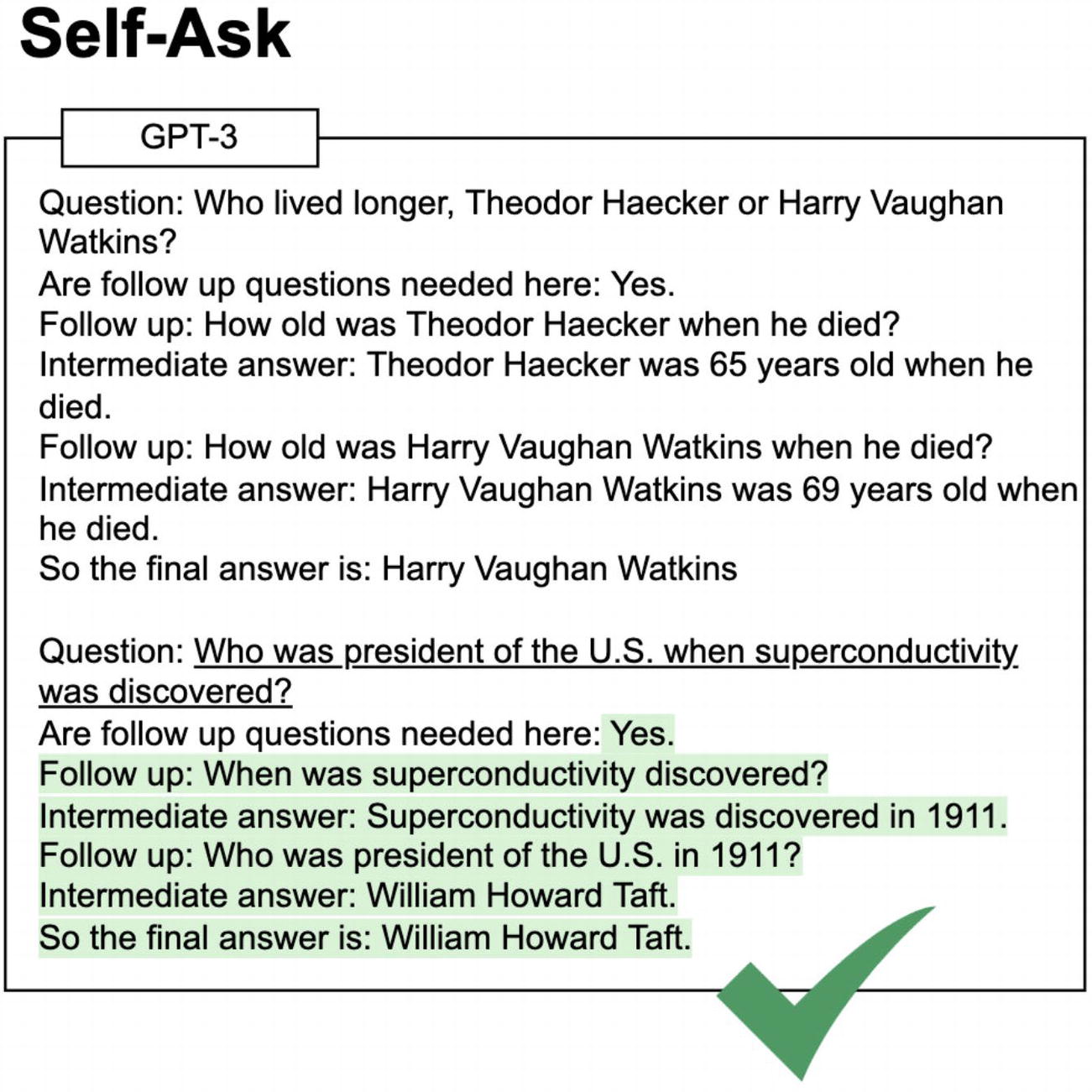
**self-ask-with-search**

This agent is based on another proposed method to improve an LLM’s reasoning and logic abilities. This method is called self-ask, and you can read the paper here: <https://ofir.io/self-ask.pdf>.

The concept is to get the model to ask itself a series of questions, answer those, and repeat until the final answer is reached.

This can involve giving an example of self-asking in the prompt.

This is shown in Figure [3-3](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_3_Chapter.xhtml#Fig3).



***Figure 3-3***

Self-ask examples from the original self-ask paper

In the paper, one of the enhancements on self-ask was to include a search engine. From the paper:

“self-ask clearly demarcates the beginning and end of every sub-question.

Therefore, we can use a search engine to answer the sub-questions instead of the LM. Search engines have features that LMs lack, such as an ability to be easily and quickly updated”

This just means a search tool allows the LLM to have access to more recent, up-to-date information as well as access to information retrieval algorithms and abilities under the search API/engine.

Okay, now that you’ve learned about agents, let’s get to actually making one yourself.

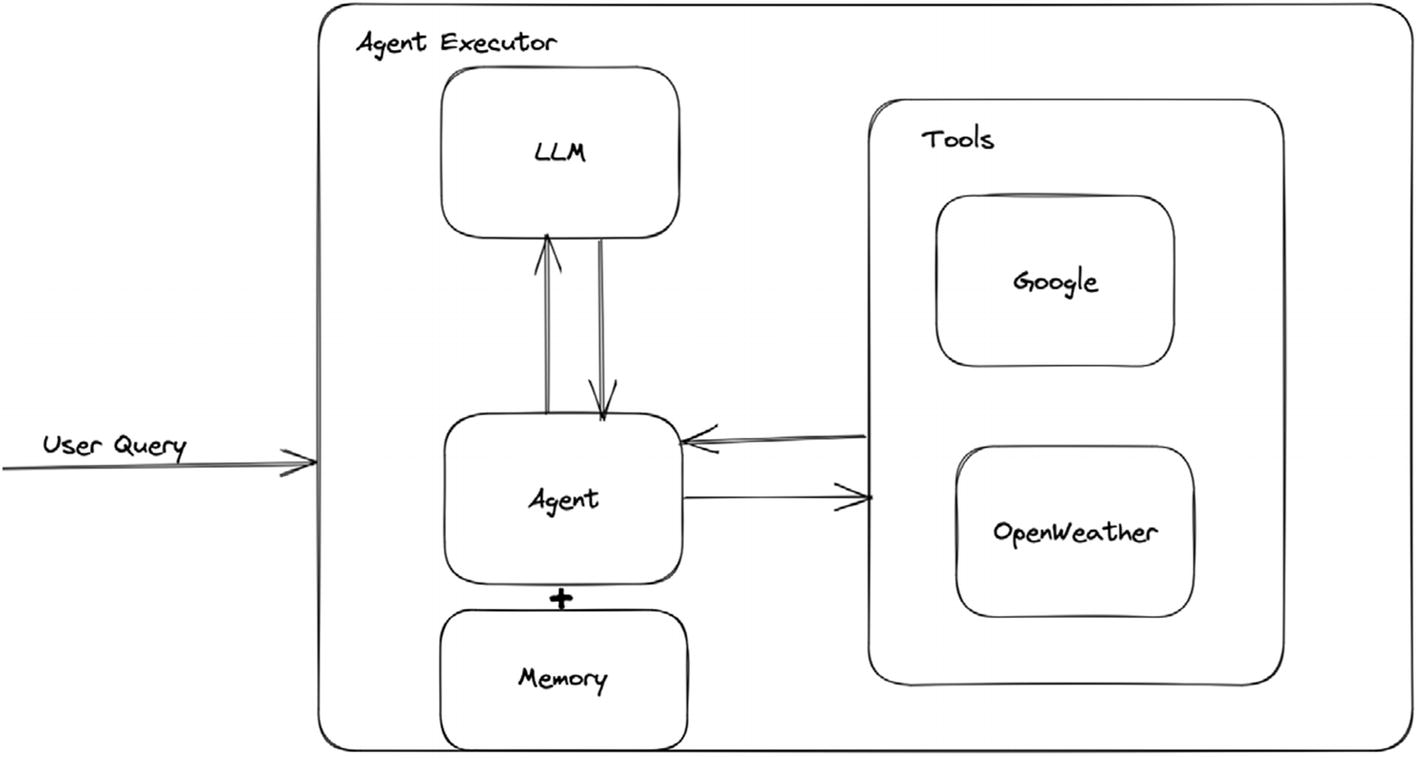
**The App**

Okay so the app you’re going to build is a day planner for any given city. It’ll be able to take into account the weather, understand what kind of activities you want, and give you tailored recommendations.

For that, you’ll need two tools to start with:

* Weather
  + Specifically, OpenWeather, but if you wanted to, you could also use another API and write a custom tool.
* Up-to-date info about places in a city
  + Specifically Google API, but again you can use another one

So with these tools in mind, take a look at Figure [3-4](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_3_Chapter.xhtml#Fig4) for the overall setup of your agent. It’s going to have memory and access to an array of tools, and agent executor will help orchestrate.



***Figure 3-4***

Reason + Act examples from the original ReAct paper

Okay, now the actual libraries I’m going to use:

* For the UI: Streamlit – There are others for you to try as well.
  + Gradio and Chainlit are the other two most popular ones.
* LangChain built-in tools:
  + OpenWeatherMapAPIWrapper
  + GoogleSerperAPIWrapper
    - This is a wrapper around serper.dev, which gives me access to Google search (and other APIs if desired).
    - LangChain also has wrapper for direct Google API access such as GoogleSearchAPIWrapper or GooglePlacesAPIWrapper.
* Agent Type:
  + I’m going to use a conversational-react-description agent, because I want both ReAct and a memory.

On to the code, I won’t go through the entire code base; you can see that on GitHub, just the parts of note.

In the first code snippet, you’re setting up your tools. You instantiate them and pass them into a Tool object, with a description and which function should be run. This is what tells the LLM what each tool can be used for – and allows the LLM to make the decisions. The agent executor uses whatever is in the func field, to actually execute, when the LLM makes a decision.

tools = [

   Tool(

       name="Search",

       func=search.run,

       description="Useful for when you need to get current, up to date answers."

   ),

   Tool(

       name="Weather",

       func=weather.run,

       description="Useful for when you need to get the current weather in a location."

   )

]

And then you set up the memory (recall, you did this in Chapter [2](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_2_Chapter.xhtml)) as shown here:

memory = ConversationBufferMemory(memory\_key="chat\_history")

Then you set up an LLM chain; recall from the chain section, this chain is one of the most simple chains, and all it does is make the call to the LLM and get the output.

llm\_chain = LLMChain(

   llm=ChatOpenAI(

       temperature=0.8, model\_name="gpt-4"

   ),

   prompt=prompt,

)

Also, take note here, you can replace the LLM field with an LLM of your choice. I’m using the wrapper for GPT-4 – but LangChain has wrappers for many others.

And then you set up your agent; this is where you pass in any chains, tools, and memory. Take note here of max\_iterations. I’ve set this to 3 because the ReAct framework technical could go on for almost an infinite number of loops for more curly queries. And even for less complex ones, there is a chance it could loop through many, many times, and since each loop costs money (i.e., a call to an API), I recommend locking down the number of iterations. Even for a self-hosted model, locking down iterations is a good idea depending on your use case; otherwise, the agent might take just way too long to come up with an answer for you.

agent = ConversationalAgent(llm\_chain=llm\_chain, tools=tools, verbose=True, memory=memory, max\_iterations=3)

Finally, you set up the agent executor that takes in the agent, tools, and optionally callbacks.

agent\_chain = AgentExecutor.from\_agent\_and\_tools(

   agent=agent, tools=tools, verbose=True, memory=memory

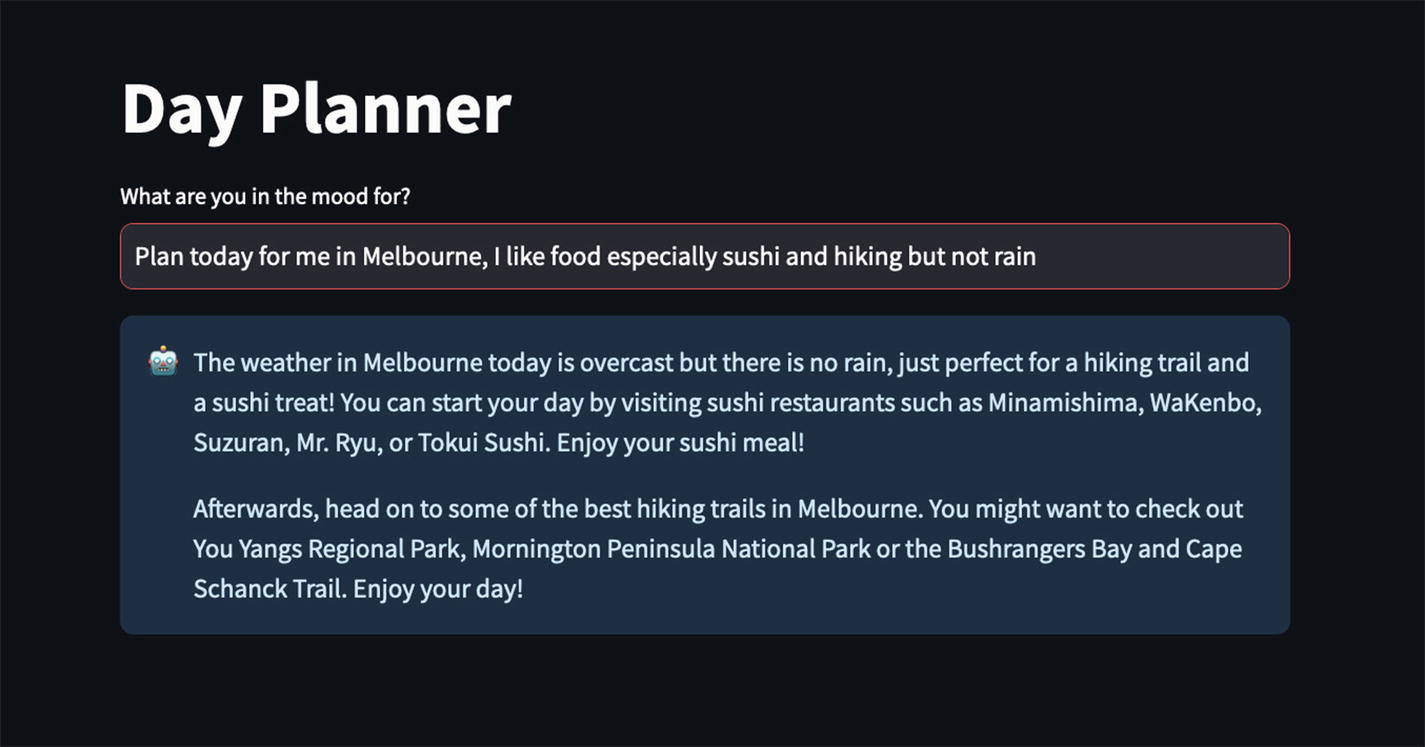
)

Now, if you go ahead and run your application with

streamlit run day\_planner\_agent.py

it’ll take you to your nice UI, where you can start querying it.

First, I put in my request about Melbourne, food, hiking, and not liking rain. Notice how the response includes info on the weather and sushi places and hiking places.



***Figure 3-5***

UI and Input + Output for your new Day Planning Agent

Then in Figure [3-6](https://learning.oreilly.com/library/view/building-generative-ai-powered/9798868802058/html/610509_1_En_3_Chapter.xhtml#Fig6), you can see the exact ReAct framework being executed. The main concept being first a thought such as “Do I need to use a tool?” then an action either use a tool or no tool and get an answer. Then an observation based on the output of the action taken. Then a thought, then action, then observation, and so on, until you get a final answer.



***Figure 3-6***

Reason + Act for your own agent

And there you have it, you have an agent that can reason and access the outside world.

The next steps for you would be to take a look at LangChain and see what kind of agents you would like to build. In this example, it has been a chat interface, but you can decide on the kind you want, maybe no interface, maybe it just runs continuously in the cloud somewhere.

**Summary**

In this chapter, you built your first agent that had reasoning abilities and access to the external world. You learned the next building blocks in LangChain: chains, tools, and agents. With this knowledge, you can start building some more complicated agents to do certain tasks for you. Remember, everything in LangChain is plug and play, so try experimenting and plugging in new libraries and tools.