FROM SCRATCH

BUILD A Multi-Agent System

With MCP and A2A

Val Andrei Fajardo

MEAP

MANNING

Build a Multi-Agent System (from Scratch)

welcome

Thank you for purchasing the MEAP for *Build a Multi-Agent System (From Scratch)*.

Multi-agent systems and the LLM agents that comprise them are some of the most discussed and worked on topics in AI today. Everyday tasks like searching the web, as well as more complex tasks like building entire codebases of software applications, are some examples where LLM agents have already been deployed. While they are by no means perfect systems, many recognize the great potential for agent-to-agent interactions to reshape the way many tasks are done in our society today.

There are more than a handful of LLM agent frameworks that exist today, which I have used extensively and even contributed to building. In fact, assuming some prior familiarity with LLMs and having past experiences programming with Python or JavaScript/TypeScript, you can probably already create LLM agents as well as multi-agent systems and have them autonomously perform tasks with any one of these frameworks.

This book, however, employs a hands-on approach to help you gain a deeper understanding of the inner workings of a multi-agent system and the LLM agents that comprise them by having you build these from scratch. This is not so different from how you might deepen your knowledge in LLMs by learning how to implement attention and transformers yourself.

To build a multi-agent system from scratch, our journey starts with the building of a foundational LLM agent. We'll also incorporate a few significant enhancements to it, such as making it MCP-ready. The vast network of tools and resources made available through MCP increases the potential of LLM agents that can leverage them. It also helps to offload some of the responsibilities for developing robust tooling for LLM agents to external teams and organizations. We'll also consider how to implement human-in-the-loop capabilities and memory for LLM agents, before finally

taking on the step of assembling multi-agent systems through the Agent2Agent protocol.

We'll package all our code, which includes the required infrastructure, such as interfaces for tools and LLMs, in the book's very own LLM agent framework that you'll get to develop for yourself.

This framework is primarily designed for educational purposes, rather than being deployed in production settings. Nevertheless, it will give you the foundation to work more confidently and effectively with any other LLM agent and multi-agent frameworks of your choosing or even to develop your own specialized solutions.

Please be sure to post any questions, comments, or suggestions you have about the book in the liveBook discussion forum.

Val Andrei Fajardo, PhD

In this book

1 What are LLM Agents and Multi-Agent Systems?

This chapter covers

- Current real-world applications of LLM agents and multi-agent systems
- What LLM agents are and why LLMs alone are insufficient
- Important design patterns, enhancements, and protocols for LLM agents
- When applications may benefit from multi-agent systems
- A roadmap for developing LLM agents and multi-agent systems

If a user asks a Large Language Model (LLM) where to find the best value in croissants in New York City, the LLM might respond, ‚ÄúI will search the web for highly-rated croissants and their prices.‚Äù LLMs are very good at expressing intent to act toward a specific goal‚Äîto generate text that tells us what they are going to do to resolve a query. At this point, however, we run into a critical limitation: since LLMs are only text-generators, they cannot act on their intentions. They can articulate a plan for processing a task, but cannot carry it out‚Äîunless they are surrounded by a system to orchestrate the plan and execute the actions.

These orchestration systems are called *LLM agents*. We‚Äôll add some nuance to this definition soon, but for our purposes, LLM agents are systems that automatically turn the LLM‚Äôs intentions into actions.

LLM agents work by interfacing with a key capability of modern LLMs: tool-calling. By *tool-calling,* we mean we can give the LLM (in text) a list of tools and a description of what those tools do, and the LLM can generate (in text) a tool-call request to call on the appropriate tool to carry out its intent when queried. The actual processing of this request occurs elsewhere in the application, and its results are sent back to the LLM for synthesis and response.

LLM agents utilize tool-calling extensively when performing tasks for users. They ensure that the tool-call requests made by the LLMs are processed, and results are sent back to the LLM. Without an LLM agent, users would have to manage this back-and-forth between tool processing and querying the LLM themselves.

As in real life, the more tools in your toolbox, the more you can do; equipping an LLM with tools for web search, math calculations, and code interpreters, for instance, makes it capable of handling a variety of tasks. We can build our own tools for LLMs to use, but we can also rely on the tools that others have created. Anthropic,Äôs Model Context Protocol (MCP) is a popular standard for how LLM agents access third-party tools with which they can equip their underlying LLM. Many tools, as well as other resources, are made available through MCP, and tapping into them dramatically increases an LLM agent,Äôs potential.

It is also possible to combine multiple LLM agents into a single system in order to improve task performance. We refer to such systems as *multi-agent systems* (MAS). In MAS, individual LLM agents collaborate with each other to perform tasks on behalf of a user. Google,Äôs Agent2Agent (A2A) protocol helps facilitate these collaborations by defining a standard for agent-to-agent interactions. With A2A, even LLM agents built using different frameworks can collaborate with each other to accomplish a task.

To work with LLM agents and multi-agent systems, and make the most of them, it,Äôs important to have a deep understanding of how they work. To gain that understanding, we will

build our very own LLM agents and multi-agent systems from scratch. We will build the complete infrastructure for LLM agents and MAS from the ground up, including interfaces for LLMs, tools, MCP resources, and A2A connectivity. And we,Äôll package all of these into our very own LLM agent framework.

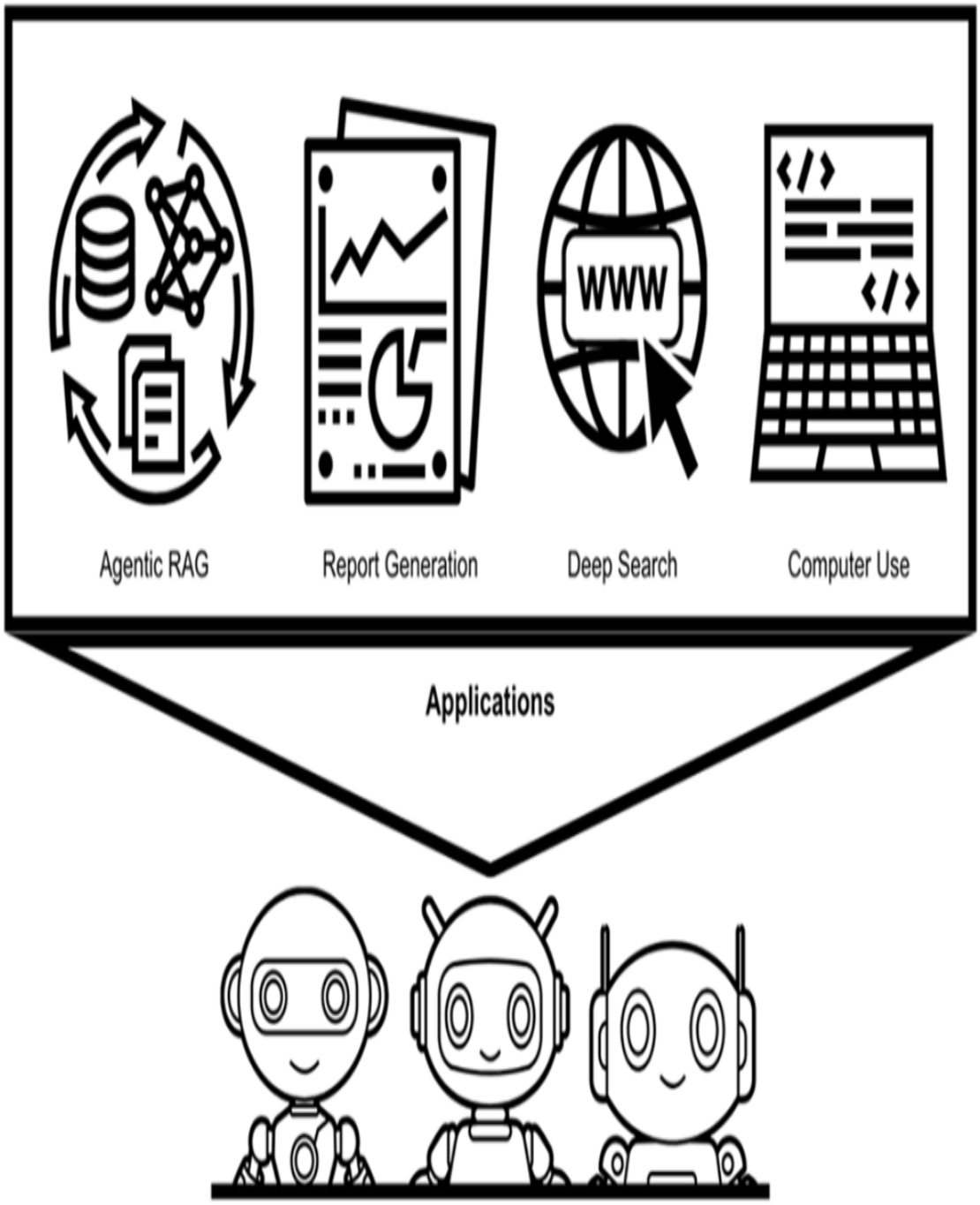
Several LLM agent frameworks already exist, and most AI engineers and practitioners use one of them. But the goal of this book is to give you a deeper understanding of how these LLM agents function, so you can work confidently and efficiently with these existing frameworks or build

specialized solutions specific to your needs. That‚Äôs why we are building ours from scratch.

1.1 Where LLM Agents and Multi-Agent Systems are useful

To get an idea of the broad usefulness of LLM agents and MAS, let‚Äôs discuss a few real-world use cases, some of which are depicted in figure 1.1.

Figure 1.1 The applications for LLM agents are many, including agentic RAG, report generation, deep search and computer use, all of which can benefit from MAS.



1.1.1 Report generation

The typical process for producing a report involves the collection of a large body of information, synthesizing it into key insights, and summarizing these insights into a structured output format. Since LLMs can synthesize large bodies of text information, the report generation task has become a

popular task for LLM agents. For instance, we might task an LLM agent to collect statistics on a variety of investment opportunities, and then provide a risk-and-opportunity assessment report for each of them. Note: reports like these can be marred by LLM hallucinations, so monitoring is crucial.

1.1.2 Web search and deep search

Performing searches against general user queries is a common use for LLM agents. Here, the LLM agents feature an LLM that has likely been fine-tuned for enhanced reasoning capabilities and web search tools. Perplexity.ai is one of the new web search engine companies that use LLM agents, but OpenAI,Äôs ChatGPT and Anthropic,Äôs Claude web applications have also recently added web search.

An extension of web search is deep research, which involves a multi-step orchestration logic that executes steps of deep browsing of webpages, followed by synthesis and report generation steps. Google,Äôs Gemini Deep Research product employs an orchestration logic that involves planning, searching, reasoning, and reporting. The other large LLM providers offer their own versions of deep research. Later in this book, we,Äôll implement our very own deep research agent using our LLM agent build.

1.1.3 Agentic RAG

LLM agents can also be used as part of a retrieval-augmented generation (RAG) system, otherwise known as Agentic RAG systems. In these applications, LLM agents are equipped with tools for querying previously built knowledge stores that contain artifacts to help answer user queries. Imagine a company with all its meeting notes and internal documentation indexed into a set of knowledge stores. An agentic RAG system can then retrieve context from these documents to answer queries supplied by the company,Äôs employees.

1.1.4 Coding LLM agents

One of the more popular uses for LLMs and LLM agents has been for coding and software development. There,Äôs even a new term, ‚Äúvibe

coding,,Äù for giving the reins over to an LLM or LLM agent to code entire projects or applications, with the human only providing natural language instructions. Coding LLM agents can also work together to contribute to an application,Äôs code base, very much like how human software developer teams contribute to a project. In these applications, LLM agents can be equipped with sandboxed code interpreters for executing arbitrary code in secure environments.

1.1.5 Computer use

LLM agents have also recently been equipped with tools that increase their scope and permissions, including those providing controls to entire applications, such as web browsers. With computer-use applications, the LLM agents can even be given control of the entirety of the operating software. With these kinds of permissions, LLM agents can perform tasks such as ordering food, buying concert tickets, and more, all through using a computer, much like how humans would. In this way, LLM agents can be viewed as next-generation Robot Process Automation (RPA) systems, which traditionally are rules-based and cannot flexibly adapt or apply reasoning to make decisions like LLM agents can.

1.1.6 Enhancing applications with MAS

All of these uses for LLM agents may be enhanced through MAS by using specialized agents across different components of an overall task. In the report generation application, you might have a specialized LLM agent that can more effectively summarize or extract information from domain-specific datasets, like financial documents. This agent might collaborate with another LLM agent that is responsible for producing well-structured reports. Or we might employ a team of coding LLM agents for building a full-stack application: you might have one that builds front-end code and another that creates the backend. In principle, MAS excel when complex tasks can be decomposed into smaller sub-tasks, where focused LLM agents outperform general-purpose ones.

1.2 What is an LLM agent?

A simple definition of an agent is some entity that acts autonomously to perform a task. Since LLMs can only generate text and cannot act, they cannot be viewed as agents.

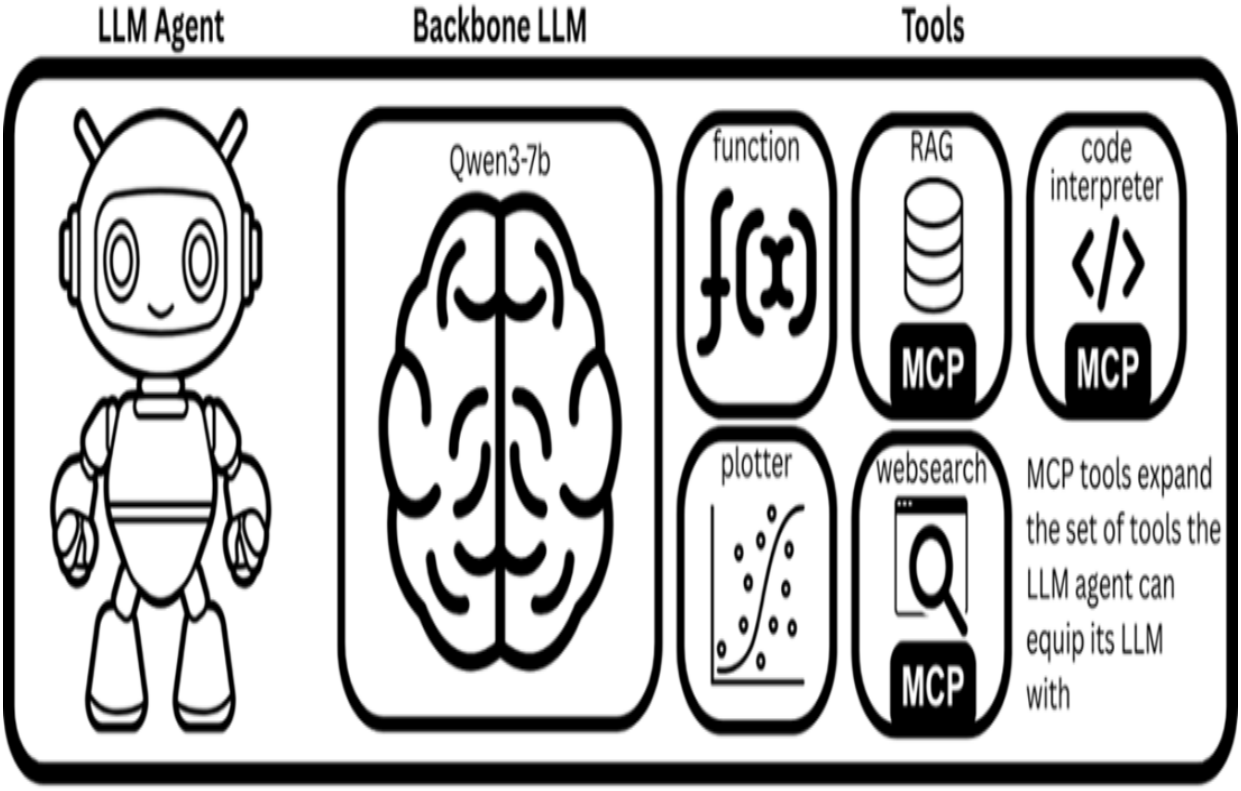
As mentioned, LLMs can, however, express intent to act through text. They can generate tool call requests and can formulate plans to perform tasks. So if we build a system around an LLM that can orchestrate the executions of these generated tool-call requests and plans, that system could be viewed as an agent.

Definition

An LLM agent is an autonomous system, comprised of a backbone LLM and tools, that acts on tool-call requests and plans formulated by the LLM to perform tasks on behalf of a user.

Figure 1.2 shows an LLM agent that utilizes the Qwen3-7b (i.e., the 7 billion parameter version of Qwen3) model as its backbone LLM and is equipped with five tools. Three of these tools are accessed through Anthopic‚Äôs MCP, meaning third-party tool providers could have created them.

Figure 1.2 An LLM agent is comprised of a backbone LLM and its equipped tools.



1.2.1 Prerequisite LLM capabilities

By our definition, an LLM agent depends on the backbone LLM‚Äôs plans and requests for tool calls to accomplish tasks. To be effective, LLM agents require their backbone LLM to make sensible choices for the next actions, including which tool calls to perform, after synthesizing the results of previous actions. Two LLM capabilities that can meet this requirement are planning and tool calling.

Planning

LLM agents are often implemented so that their backbone LLMs create initial plans to execute a given task. This happens at the very beginning of the task execution and clearly relies on the overall capability of the backbone LLM to formulate plans. Setting an incorrect plan at the very beginning of the task execution can lead to catastrophic outcomes such as incorrect task results, task execution timeouts, or massive inefficiencies.

Suppose we were to execute on our task of finding the best-value croissants in New York City. The LLM agent might first receive the original user request and formulate an initial plan. It does this through text generation, of course, and could look something like the following:

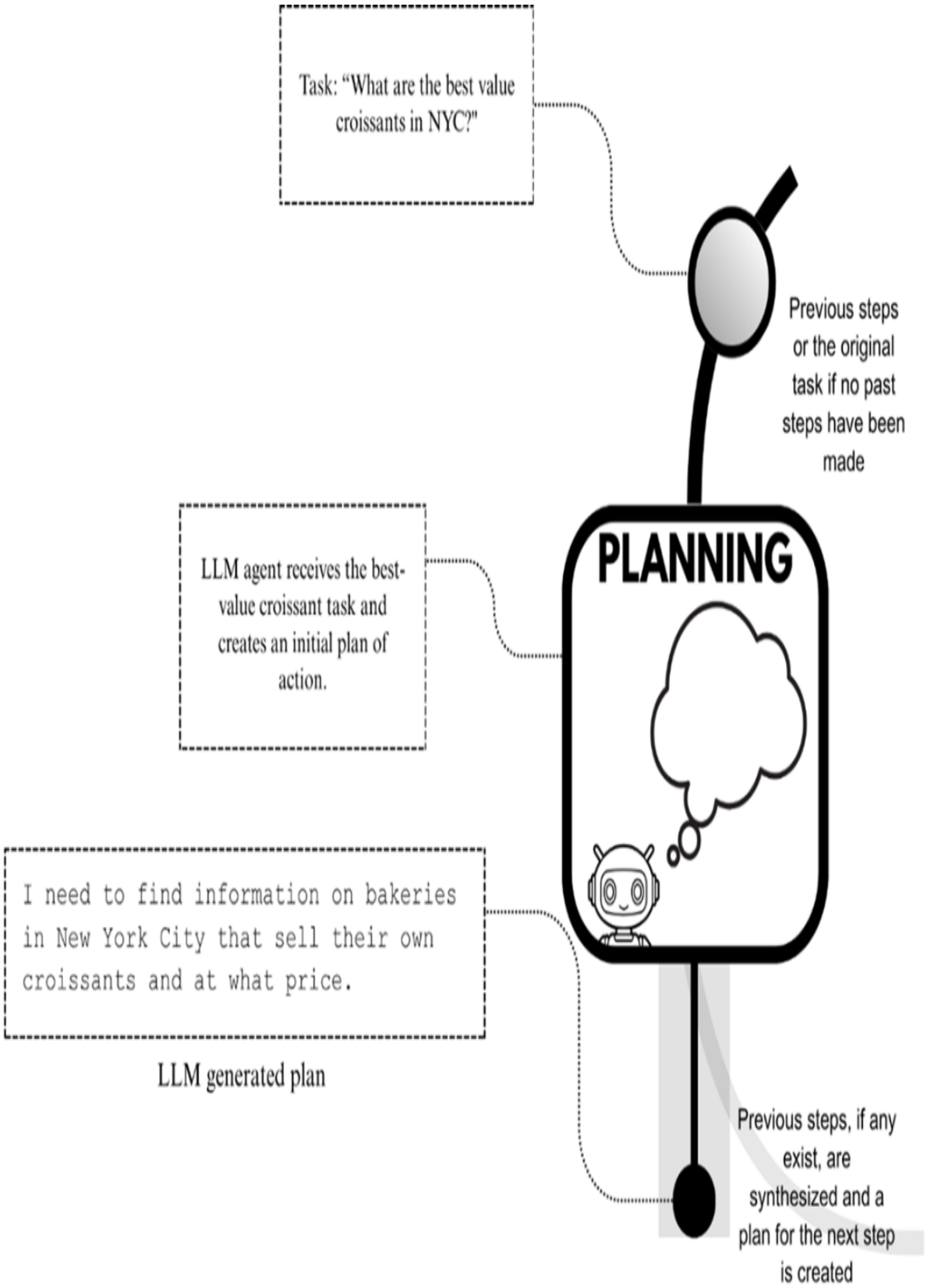
```
"I need to find all of the bakeries in New York City that sell
croissants and check their prices as well as their ratings.
Then, I need to build an analysis with this information to
determine the best-value croissants in NYC."
```

This initial plan typically forms the start of a new sub-step execution, which is part of the broader task processing. As you can imagine, planning is not only used at the beginning of the task execution, but also throughout its entirety.

One common way to implement a task execution is through a processing loop that produces steps or actions towards task completion. We utilize the LLM‚Äôs planning capability to synthesize the results of the previous steps or actions and their contribution to the overall task execution. It is here that the LLM agent, through its backbone LLM, can adapt its current plan to a new one based on these past results. The LLM agent could, for example, course-correct if it deems that the past steps have led the execution down a not-so-happy path, or it could determine that the task has been completed, perhaps earlier than anticipated, and exit the task execution with the appropriate task result.

Figure 1.3 shows how LLM agents plan with LLMs.

Figure 1.3 LLM agents utilize the planning capability of backbone LLMs to formulate initial plans for tasks, as well as to adapt current plans based on the results of past steps or actions taken towards task completion.


Task: "What are the best value croissants in NYC?"
Previous steps or the original task if no past steps have been made
PLANNING
LLM agent receives the best-value croissant task and creates an initial plan of action.
I need to find information on bakeries in New York City that sell their own croissants and at what price.
LLM generated plan
Previous steps, if any exist, are synthesized and a plan for the next step is created

Planning vs Chain-of-Thought and Reasoning LLMs

A related concept to planning is Chain-of-Thought, an LLM-prompting technique that attempts to have LLMs provide their reasoning and deduction steps, in addition to the final response. Prompts used to elicit these chains of thought from LLMs to answer questions could include demonstrative examples, known as few-shot exemplars.

For instance, ‚ÄúThe number 77 is divisible by 7 and 11. Thus, 77 is not a prime number‚Äù would be an exemplar answer to the question ‚ÄúIs 77 a prime number?‚Äù that could be included under an examples section of a prompt.

These prompts also typically include words like ‚Äúshow your or work‚Äù or ‚Äúlet‚Äôs solve this step-by-step‚Äù in the system-context setting portion of the prompt.

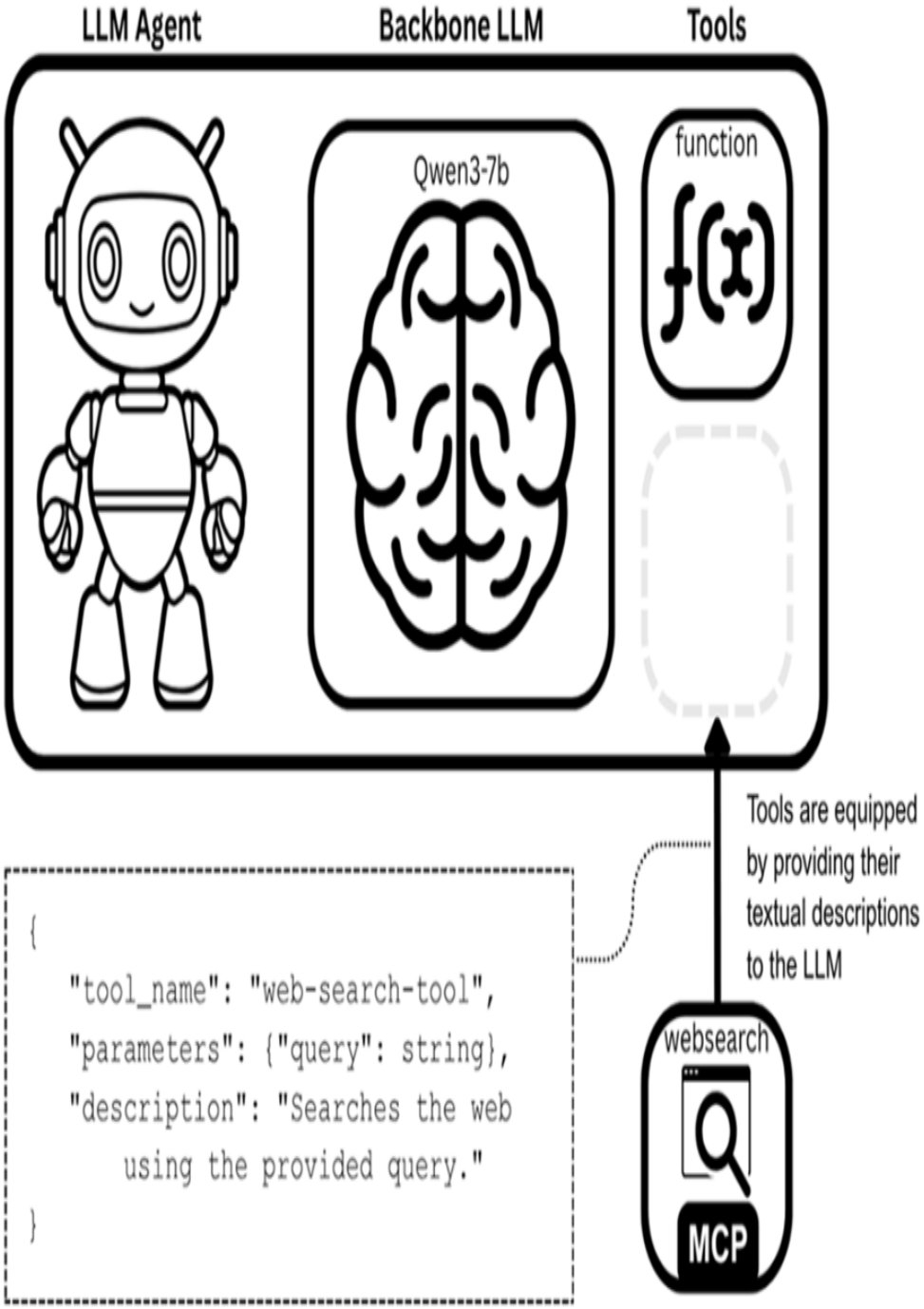
In recent times, LLMs have been further trained to generate long chains of thought, and such LLMs have come to be known as ‚Äúreasoning LLMs.‚Äù These generated sequences of text are often included in the final text output, typically enclosed in a section marked by the tags ‚Äú<thinking>‚Äù and ‚Äú</thinking>‚Äù.

As a result of their post-training, reasoning LLMs may also have increased abilities to plan, and could therefore be a worthy candidate as the backbone LLM for LLM agents.

Tool Calling

The second prerequisite capability for the backbone LLM is tool calling. Earlier, we discussed how to equip an LLM with a tool by providing it with textual descriptions of the tool‚Äôs functionality and parameters. By doing so, we enable the LLM to express an intent to use the tool. You can see this tool-equipping process in figure 1.4

Figure 1.4 An illustration of the tool-equipping process, where a textual description of the tool that contains the tool‚Äôs name, description and its parameters is provided to the LLM agent.



Let‚Äôs unpack the rest of the broader tool-calling process and overall capability.

Tools that are equipped to the LLM agent can be used for future tool calls. The entire tool-call process is shown in figure 1.5

Figure 1.5 The tool-calling process, where any equipped tool can be used.

LLM agent calls the web-search tool with the query: "best croissants NYC ratings prices".


```
{
   "tool_name": "web-search-tool",
   "parameters": {
       "query": "Croissant bakeries in
           New York City and their
           prices.",
   }
}
```

LLM generated output for a web-search tool call request

LLM selects a tool and generates a tool call request

```
tool_result = web_search_tool(
    "Croissant bakeries in New York
    City and their prices."
)
```

Web search tool execution

Tool execution commences with the parameters generated by the LLM

```
Below is a tool call result:
{
   "tool_name": "web-search-tool",
   "result": [
       "hits": [
           {
              "name": "Bob's Bakery",
              "has_croissant": True,
              "price": 3.50,
              "yelp_rating": 4.80,
           },
           ...
       ],
   }
}
```

Tool execution completes

Results are sent back to the LLM

The tool call result turned into a string to be sent back to the LLM

Each tool-call process begins with an LLM making a tool-call request. This tool-call request is text generated by the LLM, which includes the tool identifier as well as the values for the required parameters of the selected tool. LLMs often make these tool-call requests through a structured output, such as a JSON format. For example, and in continuation with our task to find the best-value croissants in New York City, the LLM may generate a web-search tool-call request that looks something like the following:

```
{
    "tool_name": "web-search-tool",  #A
    "parameters": {  #B
        "query": "Croissant bakeries in New York City and their
prices.",
    }
}
```

As you can see, the tool request that the LLM makes contains not only the tool selection but also the values for the selected tool,Äôs parameters.

After an LLM has made a tool call request, the tool is executed with the provided parameters. The results of the tool call execution are then sent back to the LLM, allowing it to synthesize the information and generate an appropriate response.

Teaching LLMs how to tool-call

To provide LLMs with this tool-calling capability, often also referred to as ‚Äútool usage‚Äù, LLMs typically undergo supervised fine-tuning, a form of post-training. This training uses instruction examples that demonstrate the types of tool calls the LLM should learn to generate. The objective is to teach the LLM to adhere to the specified format for making tool call requests and to learn when tool calls would be appropriate. An LLM that cannot generate tool call requests would undoubtedly be a bad choice for the backbone LLM of an LLM agent.

Having discussed the two prerequisite capabilities of the backbone LLM, let's now see how the LLM agent uses these capabilities to perform tasks through its processing loop.

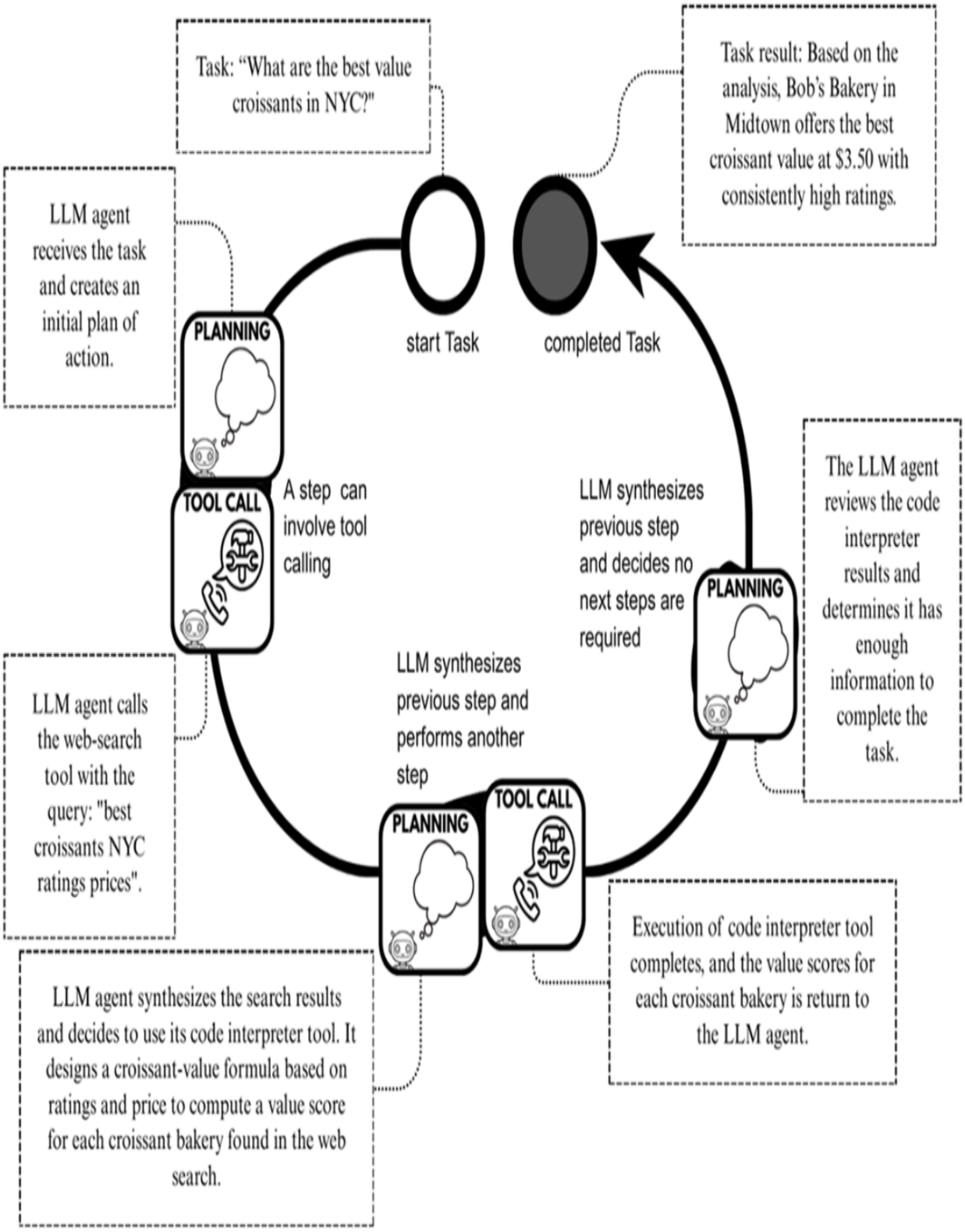
1.3 The processing loop

The processing loop of an LLM agent is where all the action takes place. Plans are developed, tool calls are made, and steps are taken to complete the task. When initial plans fail, they can be adapted to perform the task successfully. LLM planning and tool-calling capabilities are repeatedly used within a processing loop. Figure 1.6 shows the processing loop and how tasks are performed within it. Let‚Äôs walk through this mental model together.

Figure 1.6 A mental model of an LLM agent performing a task through its processing loop, where tool calling and planning are used repeatedly. The task is executed through a series of sub-steps, a typical approach for performing tasks.

Processing Loop

where tasks are performed



A processing loop is initiated when a task is submitted to the LLM agent for execution. The actual approach for executing tasks within a processing loop is a design choice. For our LLM agent framework, our approach is to execute the task through a series of sub-steps, as shown in figure 1.6.

At the start of every step, the LLM agent synthesizes the results and progress made so far on the task to create the next intermediate plan or step. For the very first step of task execution, since no prior progress has been made, the LLM agent simply uses the user,Äôs request or task instruction to formulate the initial step,Äôs plan. The LLM agent can make tool calls within any step.

For our example task of finding the best-value croissants in New York City, let,Äôs suppose that the initial plan and tool call presented in the previous sections, where we discussed the prerequisite LLM capabilities, form the first sub-step. Here,Äôs an outline of this first step, as it fits within the overall task execution.

2 Working with Tools

This chapter covers

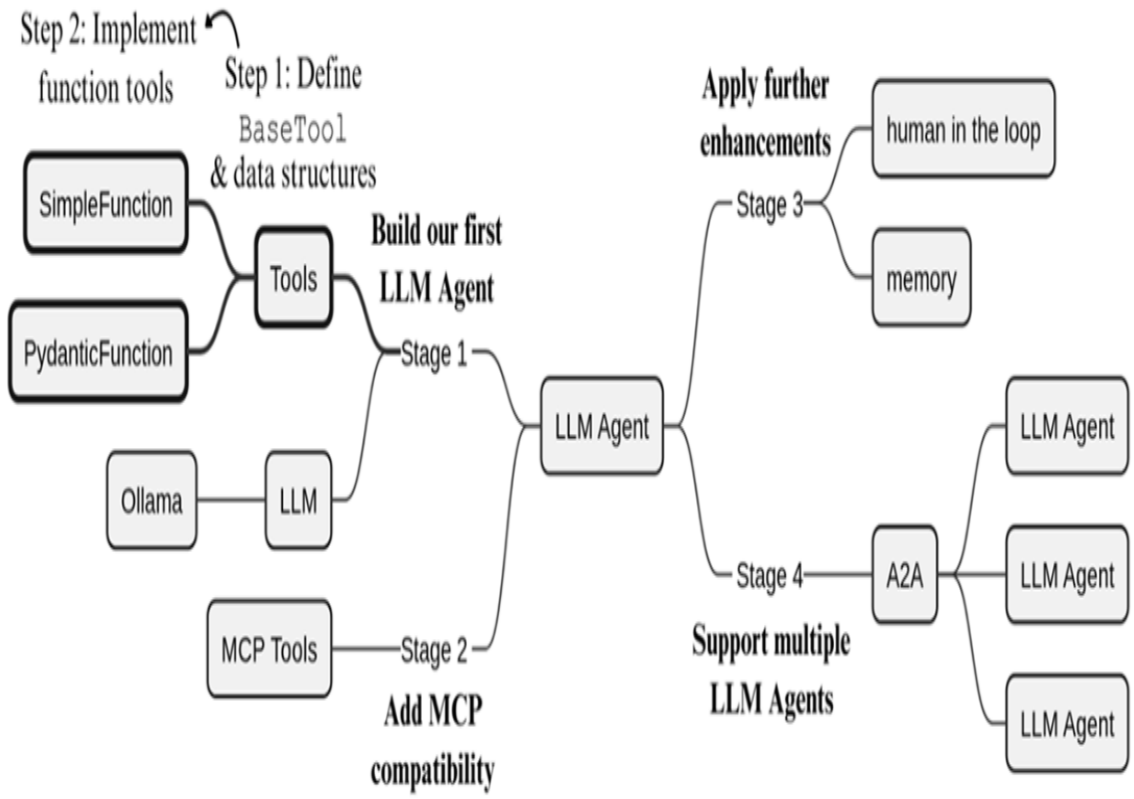
- Specifying the base class for tools to standardize how they are used in our framework
- Defining the data structures for facilitating a tool-call process
- Turning Python functions into tools to use with LLMs and LLM agents

You now know that tools are a crucial part of LLM agents. Tools, such as those for performing web searches, plotting data, and executing code in a sandboxed environment, increase the potential of LLM agents and expand the range of tasks they can perform.

Equipping an LLM with a tool requires us to provide a textual description of the tool so that the LLM can understand how to use it. LLMs can use a tool through the tool-calling process, which involves the LLM generating a tool-call request, invoking the tool, and finally returning the result to the LLM for synthesis and response.

The focus of this chapter, as illustrated in figure 2.1, is to build the required infrastructure for defining tools and how they can be used. We'll package all our code in our own LLM agent framework, called `llm-agents-from-scratch`.

Figure 2.1 The focus of the current chapter, through the lens of our build plan introduced in Chapter 1. Before we can build an LLM agent, we first need to properly define tools and how they can be used within our framework.



To be more specific, we'll define a base class interface for tools, `BaseTool`, which will serve as a blueprint for adding tools to our framework. This blueprint will also help to standardize how tool calls are performed, as well as how the textual descriptions of tools are prepared and passed to the LLM. To completely standardize the tool calling process in our framework, we'll need to introduce a couple of new data structures that represent the input and output of tool calls.

In addition to defining the base class and adding the necessary data structures, we'll also implement a couple of subclasses that will help us to create LLM tools from Python functions.

To follow along with the code examples, I recommend forking the book's GitHub repository and activating the framework's dedicated virtual environment. You can do so by running the following terminal commands while in the project's root directory.

```
uv sync  #A

# mac or linux
source .venv/bin/activate
```

```
# windows (powershell)
.venv\Scripts\activate
```

Additionally, a Jupyter notebook containing executable code for the example demonstrations in this chapter has been prepared and can be found in the book's GitHub repository: https://github.com/nerdai/llm-agents-from-scratch/blob/main/examples/ch02.ipynb. Code snippets marked with `# Included in examples/ch02.ipynb` are available in this notebook for you to run.

I recommend using `uv` to launch Jupyter Lab to ensure all the necessary packages are installed for you to run these examples. You can do this by running the following terminal command in the project's root directory (where `pyproject.toml` can be found).

```
uv run --with jupyter jupyter lab
```

2.1 BaseTool: a blueprint for tools

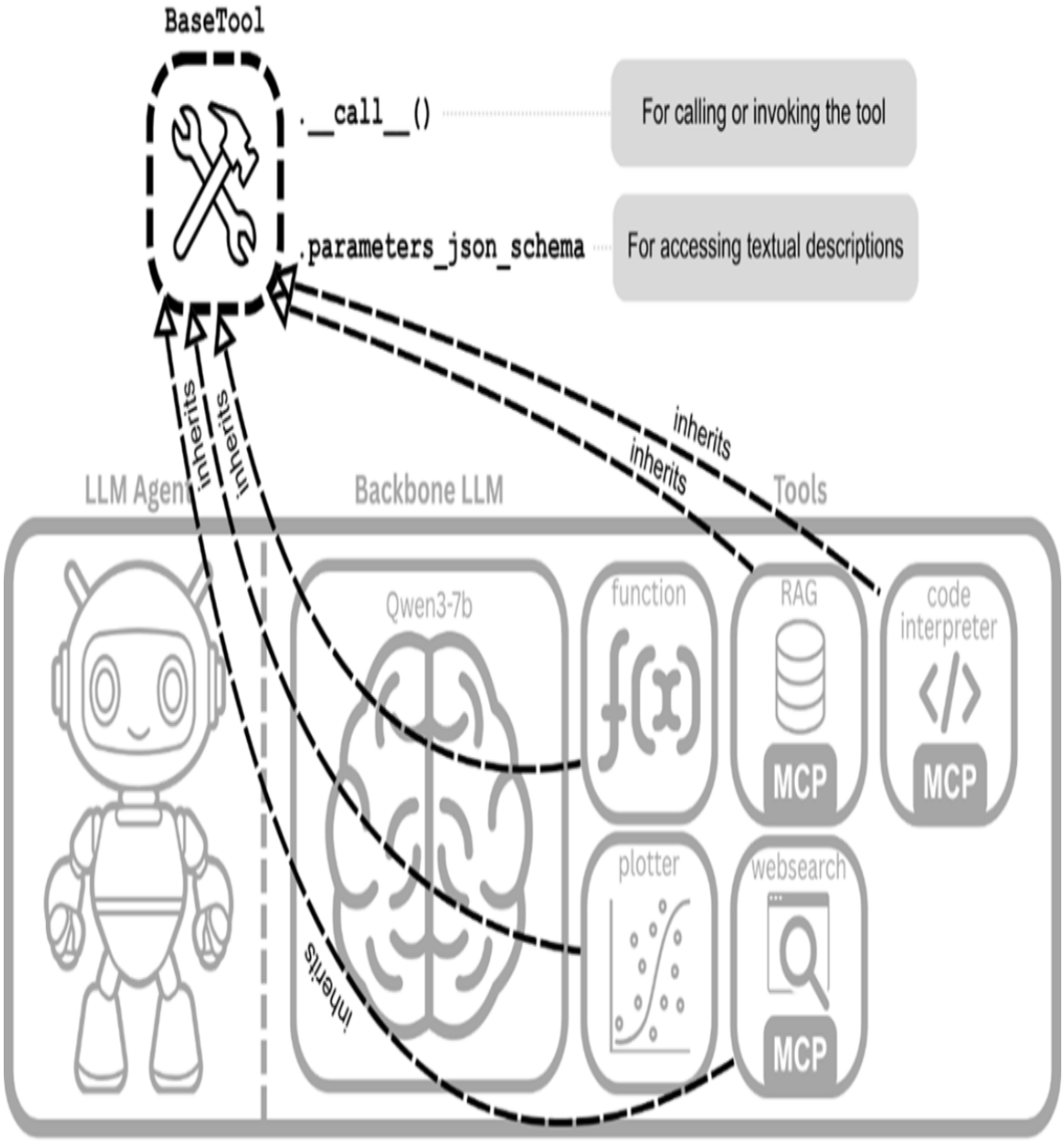
Tools can differ significantly in their functionality. For example, a tool that performs web searches is obviously very different from another tool that can plot pie charts. LLM agents with tools of various functionalities have the potential to be more potent and versatile. However, if these tools also differ significantly in how they're invoked, then it would be challenging to realize such potential. These differences might also lead to the LLM making mistakes when requesting tool calls.

A better approach would be to define a standard way for calling tools, regardless of their different functionalities. For similar reasons, we'd also want to standardize the way in which textual descriptions of tools are formatted. This standardization will pave the way for us to write reliable code that enables LLMs and LLM agents to work with these tools—as you'll see later, in Chapters 3 and 4.

We will define this standard for tools through a special base class, `BaseTool`, which we'll implement in this section. To be even more precise, though, we'll also be implementing a related base class called `AsyncBaseTool`. Together, these base classes define the standard that every tool added to our framework must conform to.

To keep things simple, let's consider only `BaseTool` for now and revisit `AsyncBaseTool` later. Figure 2.2 illustrates the same LLM agent from before, along with its tools, each of which is now shown to inherit from `BaseTool`.

Figure 2.2 Each tool inherits from the BaseTool class, allowing the LLM agent to access the textual descriptions and execute the logic of each tool in the same manner.



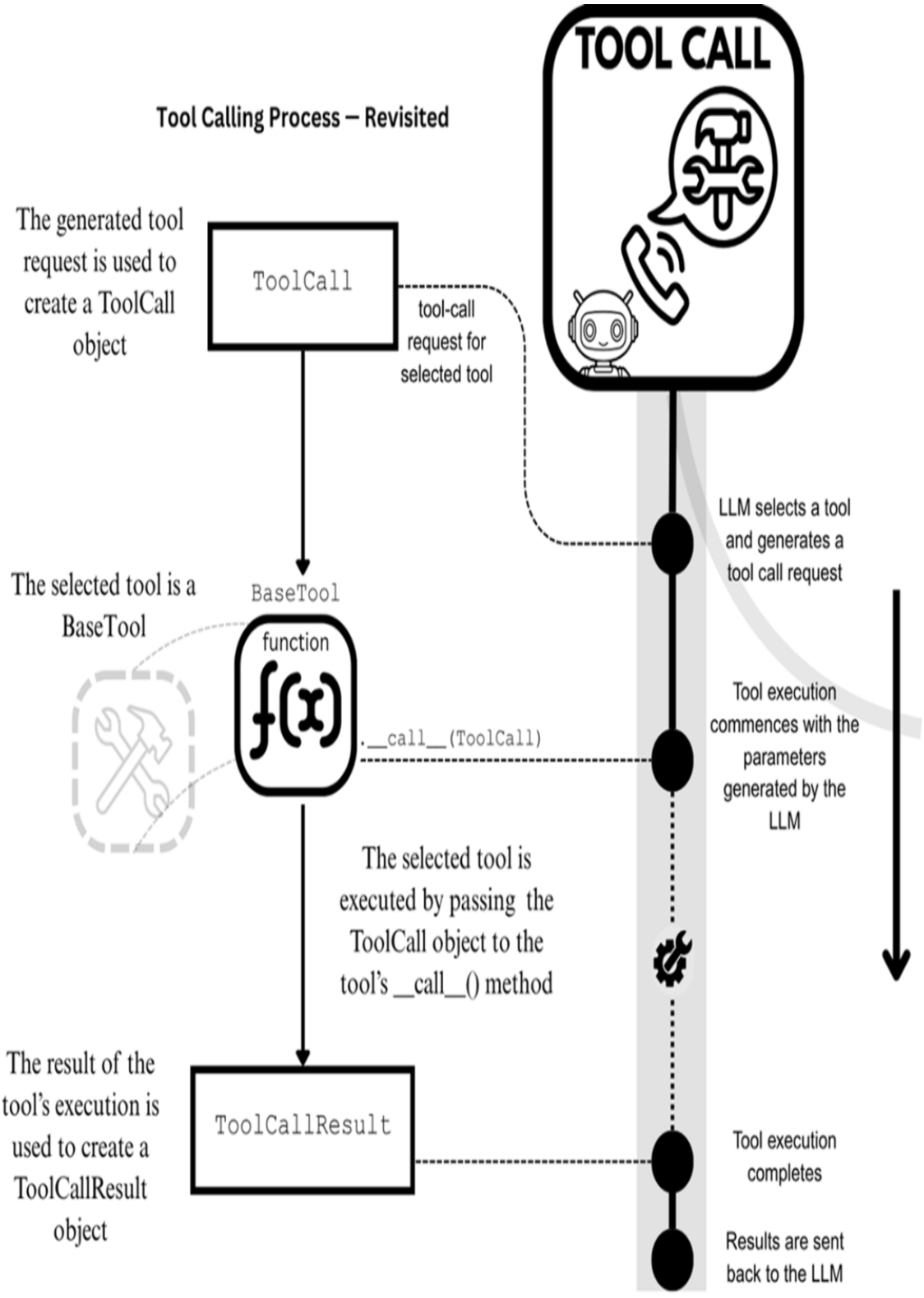
Our `BaseTool` class will specify a set of methods and attributes, which every tool that inherits from it must also support. For simplicity, figure 2.2 shows only the `__call__()` method and the `parameters_json_schema` attribute, but we will soon see the full structure of `BaseTool`.

As mentioned earlier, defining the `BaseTool` only gets us part of the way toward implementing a standardized tool-calling process in our framework. We also need

to define two new data structures for representing the input and output of a tool call. The `ToolCall` class will define our tool-call input, and the `ToolCallResult` class will define our tool-call output.

Before we write any code, let's quickly revisit the tool-calling process that we covered in the previous chapter, and see how our new classes fit into it. Figure 2.3 shows the familiar tool-calling process diagram, but with our new classes incorporated into it.

Figure 2.3 Incorporating our new classes—BaseTool, ToolCall, and ToolCallResult—into the tool-calling process, first shown in Chapter 1.

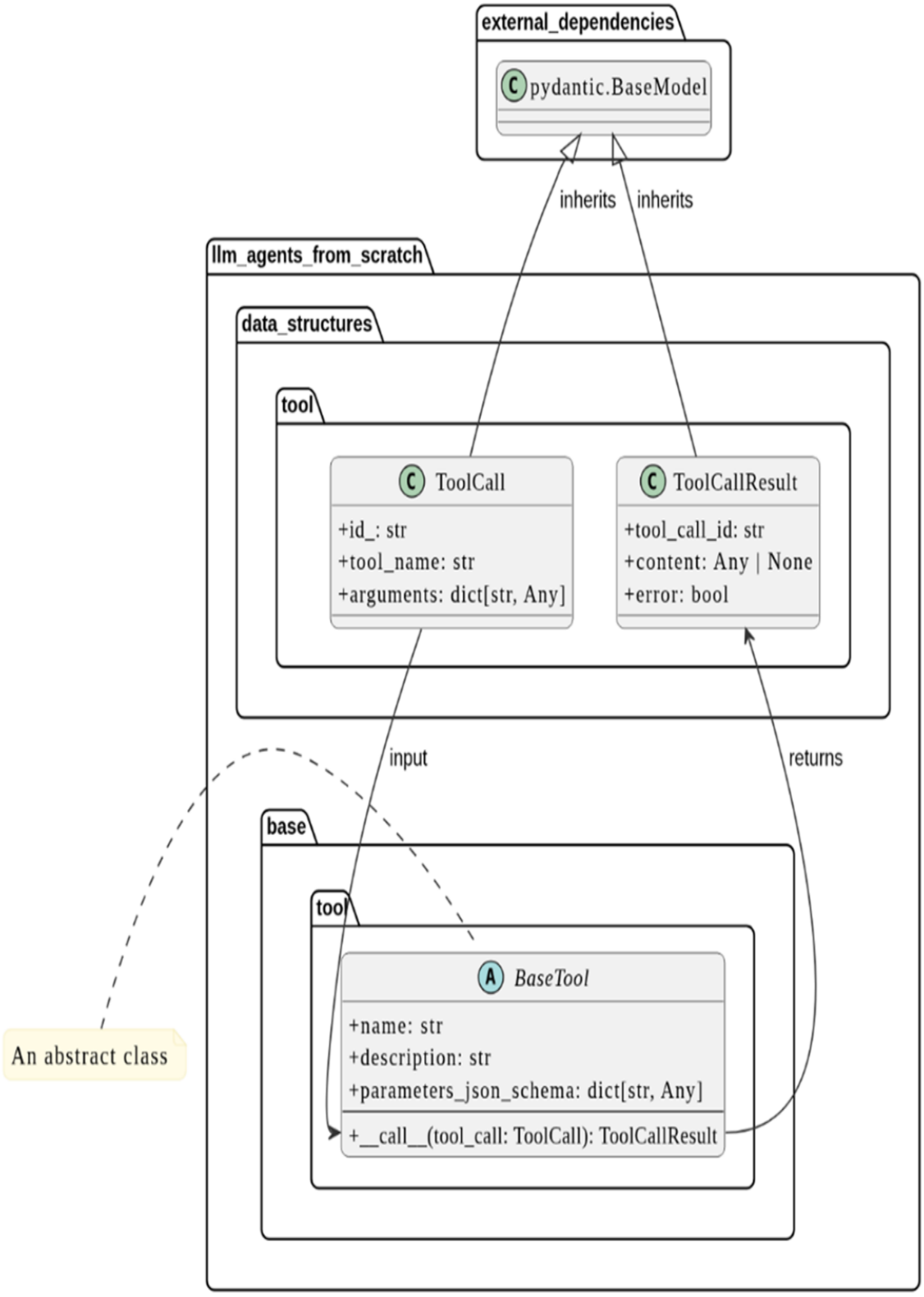


As you learned in the previous chapter, the first step to the tool-call process is for the LLM to select a tool and generate a tool-call request. We can now add more

structure to this step by packaging the generated request into a `ToolCall` object. The next step would be to execute the selected tool's logic using the parameters specified in the tool-call request. Since all tools will conform to the standard interface defined in `BaseTool`, we will be able to execute every tool's logic in the same way. That is, through the selected tool's `__call__()` method, which would accept the previously created `ToolCall` object as input. After the tool's logic is executed, the result is then packaged into a `ToolCallResult` object, which can then be passed back to the LLM.

Now that we have outlined how these new classes will work together, let's delve into their structural details, which are provided in the UML class diagrams seen in figure 2.4.

Figure 2.4 The UML class diagrams for BaseTool, ToolCall and ToolCallResult.



We covered a few of the basics for UML class diagrams in Chapter 1. You may recall from that previous discussion that attributes are outlined in the top section of

the rectangular box, while methods are provided in the bottom section. Figure 2.4 shows that `ToolCall` consists of three attributes—`id_`, `tool_name`, and `arguments`—and no methods. The `ToolCallResult` class also has three attributes and no methods. Those attributes are `tool_call_id`, `content`, and `error`. We'll go over what these attributes represent when we write the code for these two classes.

Figure 2.4 also shows the full structure of the `BaseTool` class, which we covered only partially at the beginning of this section. We were previously familiar with the `parameters_json_schema` attribute and the `__call__()` method, but now we can see two additional attributes: `name` and `description`.

Finally, figure 2.4 illustrates a couple of new UML concepts that we'll now go over. First is the inheritance relationship, which is indicated by a solid line with a hollow triangle arrowhead pointing from the child class to the parent class. You can see that both `ToolCall` and `ToolCallResult` inherit from the external `pydantic.BaseModel` class. Second, you may have noticed that, rather than the usual circle with a "C" beside the `BaseTool` text, there is a circle with an "A" instead. This "A" stands for abstract and indicates that any tool that extends the `BaseTool` class must provide implementations for the methods which have been marked as abstract; these are methods that have no default implementation in the base class. You'll see how we mark methods as abstract when we implement the `BaseTool` class.

TIP

Pydantic is a Python library that is especially useful in defining data models that require robust validation.

Now that we understand how these classes are structured and work together in the tool-calling process, let's implement them!

2.1.1 Implementing ToolCall and ToolCallResult

We'll implement our two new data structures, one at a time, starting with `ToolCall`. The three attributes of `ToolCall` were shown in figure 2.4. The `id_` attribute provides a string identifier for a `ToolCall` object, while `tool_name` and `arguments` represent the selected tool's name and the parameter values we'll use to invoke it, respectively.

The following listing shows the implementation of `ToolCall`.

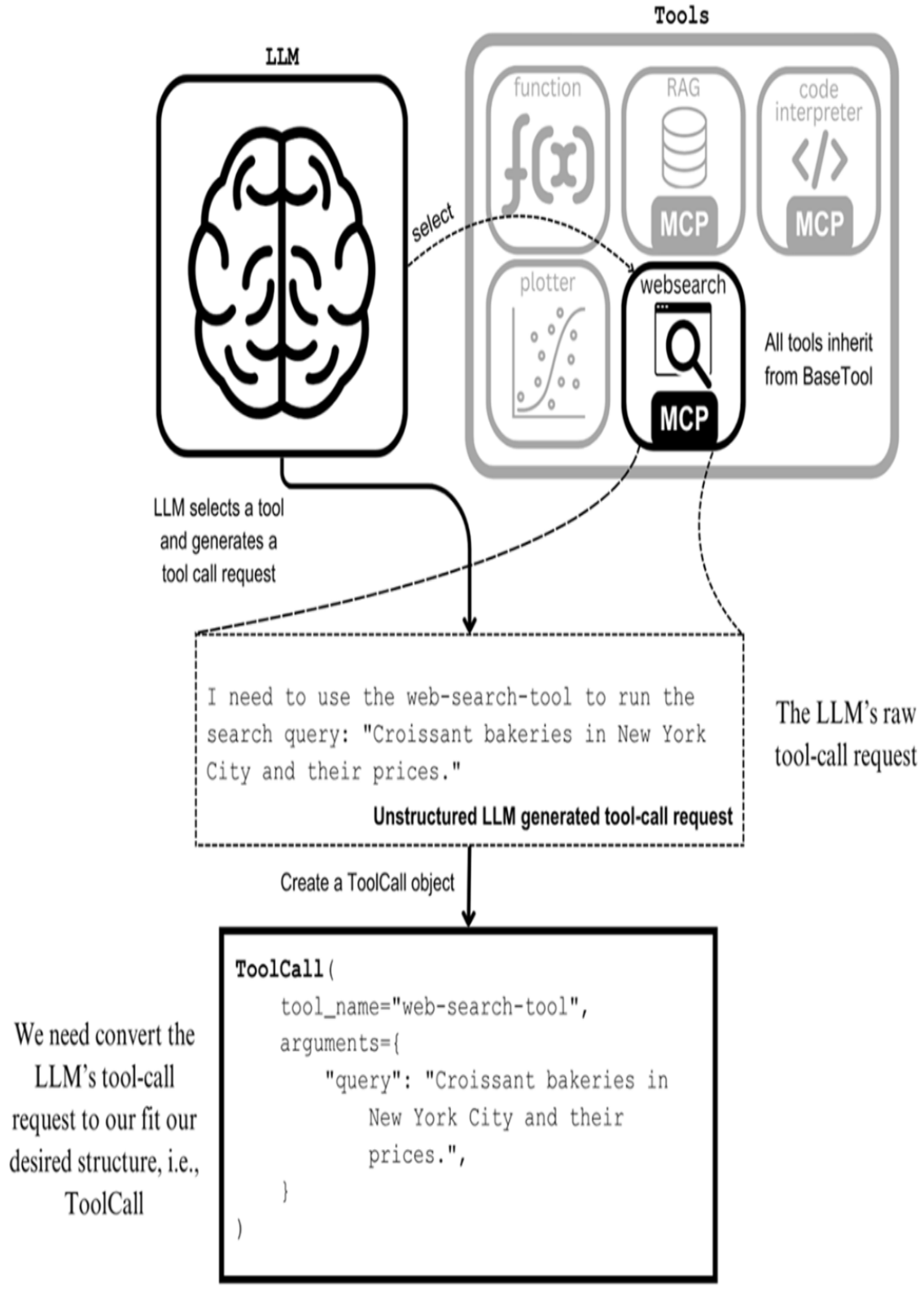
Listing 2.1 Implementing ToolCall

```
# llm_agents_from_scratch/data_structures/tool.py  #A
import uuid
from typing import Any

from pydantic import BaseModel, Field


class ToolCall(BaseModel):
    """Tool call.

    Attributes:
        id_: A string identifier for the tool call.
        tool_name: Name of tool to call.
        arguments: The arguments to pass to the tool execution.
    """

    id_: str = Field(default_factory=lambda: str(uuid.uuid4()))
    tool_name: str  #B
    arguments: dict[str, Any]  #C
```

Since `ToolCall` inherits from `pydantic.BaseModel`, we specify attributes under the class declaration. With `pydantic`, the three attributes are also called model fields. We can add customizations to any of the fields, such as providing a default factory method, as we've done for `id_`. This means that we do not need to supply a value for `id_` when creating a `ToolCall` object. I should also mention that a default constructor is provided for `pydantic.BaseModel` objects, which explains why we don't need to define an `__init__()` method ourselves.

As discussed earlier, we need to package the tool-call requests generated by LLMs into `ToolCall` objects. Let's revisit our best-value croissant in New York City example from the previous chapter. Suppose the backbone LLM generates a tool-call request in natural language text: "I need to use the web-search-tool to run the search query: 'Croissant bakeries in New York City and their prices'." Figure 2.5 shows how we create a `ToolCall` object from this request.

Figure 2.5 Creating a ToolCall object from an LLM's natural language tool-call request. Since the web-search tool is a BaseTool, we must first convert the request into a ToolCall object before invoking it.



Converting this request to a `ToolCall` object requires identifying the selected tool's name as well as the arguments we'll use to invoke it. The following code

snippet demonstrates how we can use these identified elements to create our `ToolCall` object.

```
# Included in examples/ch02.ipynb  #A
from llm_agents_from_scratch.data_structures.tool import ToolCall

croissant_tool_call = ToolCall(
    tool_name="web-search-tool",  #B
    arguments={
        "query": "Croissant bakeries in New York City and their
            prices.",  #C
    }
)
```

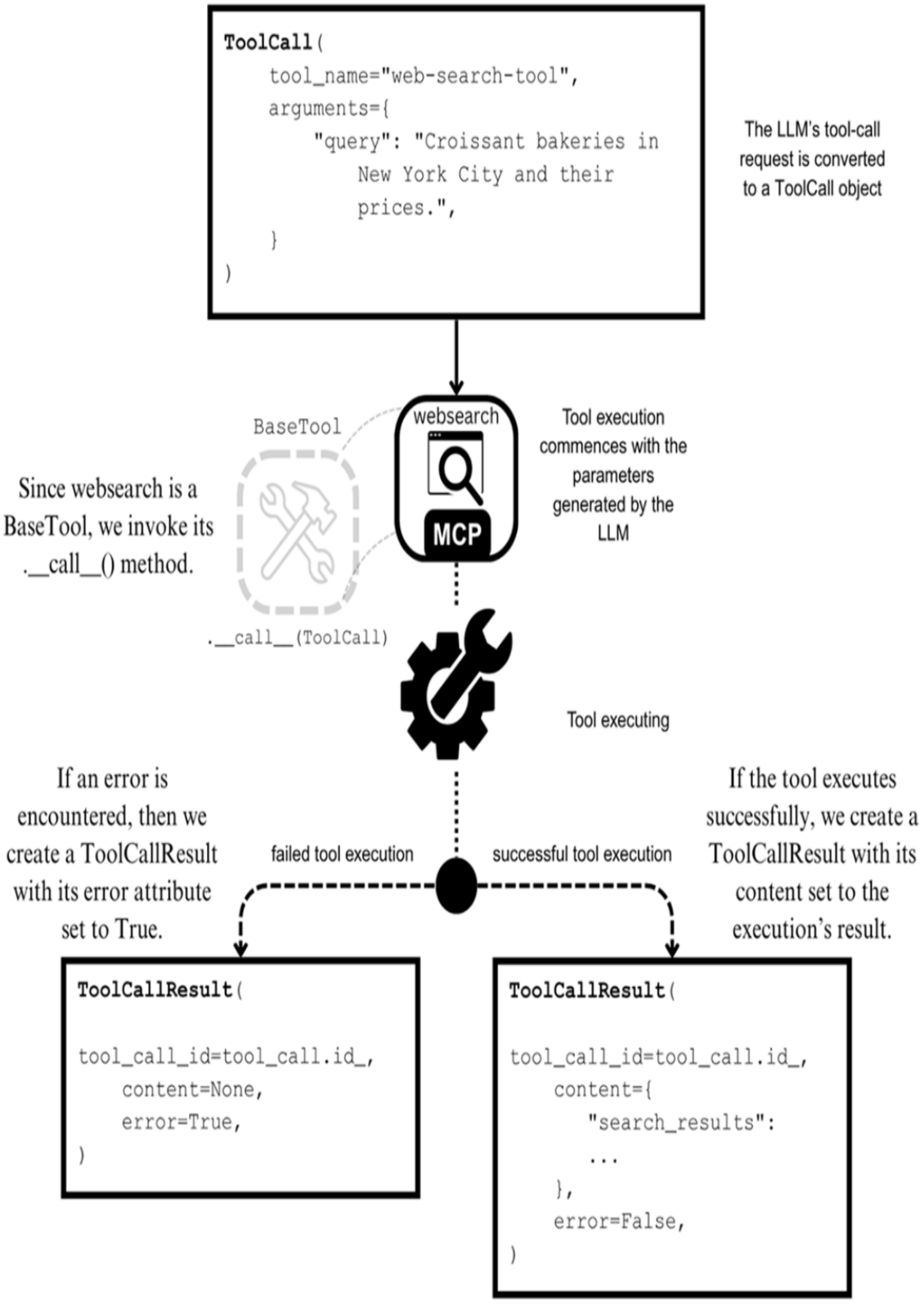
NOTE

In practice, we use APIs or Software Development Kits (SDKs) provided by LLM providers to elicit tool-call requests. In contrast to our example, these requests are typically generated according to a pre-specified structured format. This is beneficial because working with unstructured text would be brittle and pose significant challenges when extracting the required elements to build `ToolCall` objects consistently. We'll work with one such API in the next chapter when we build out the base class for LLMs.

With our `ToolCall` object created, we can now invoke the tool's `__call__()` method to execute the requested web search. Since we haven't yet implemented our `BaseTool` class, we'll have to wait a little longer before we can see this step in action.

For now, let's continue with our planned implementation of `ToolCallResult`. You saw earlier from figure 2.4 that `ToolCallResult` has three attributes: `tool_call_id`, `content`, and `error`. The `content` attribute stores the results of the tool-call execution, while the `error` attribute specifies whether or not an error was encountered. The `tool_call_id` attribute helps to tie back the result to its associated tool-call request. Figure 2.6 shows how the creation of the `ToolCallResult` object is dependent on the outcome of a tool execution.

Figure 2.6 Creating a ToolCallResult object from the selected tool's execution, handling both successful and failed executions.



After a successful execution, the result is assigned to the `content` attribute and the `error` is set to `False`. If an error is encountered during the tool's execution, the

`content` and `error` attributes are set to `None` and `True`, respectively.

The following code implements the `ToolCallResult` class.

Listing 2.2 Implementing ToolCallResult

```
# llm_agents_from_scratch/data_structures/tool.py
import uuid
from typing import Any

from pydantic import BaseModel, Field


…  #A

class ToolCallResult(BaseModel):
    """Result of a tool call execution.

    Attributes:
        tool_call_id: The id of the associated tool call.
        content: The content of tool call.
        error: Whether or not the tool call yielded an error.
    """

    tool_call_id: str  #B
    content: Any | None  #C
    error: bool = False  #D
```

As you can see, the implementation of `ToolCallResult` is similar to that of `ToolCall` since both classes inherit from `pydantic.BaseModel`.

Continuing our previous example, if we imagine a successful execution of the `web-search-tool`, we'd create a `ToolCallResult` object with its `content` set to the execution result and its `error` set to `False`, as the following code snippet demonstrates.

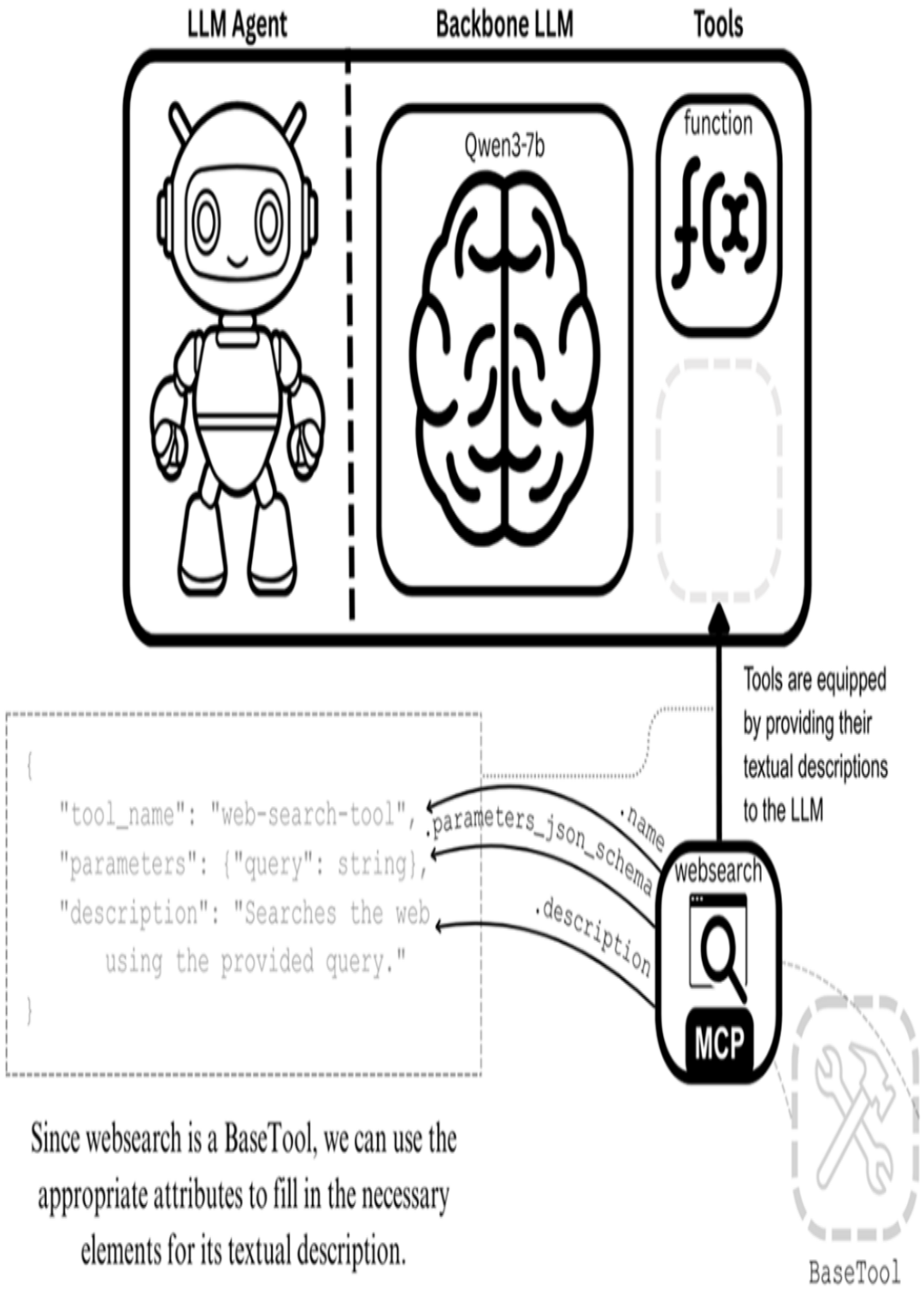
```
# Included in examples/ch02.ipynb  #A
from llm_agents_from_scratch.data_structures.tool import
ToolCallResult

result = ToolCallResult(
    tool_call_id=tool_call.id_,  #B
    content={
        "search_results": …  #C
    }
)
```

2.1.2 Implementing BaseTool

Now that we've implemented the necessary data structures for working with the `BaseTool` class, let's move on to its implementation. As we saw from the UML class diagram in figure 2.4, `BaseTool` is an abstract class with three attributes and one method.

The three attributes—`name`, `description`, and `parameters_json_schema`—provide the information needed to equip an LLM with the `BaseTool`. More specifically, they are used to prepare the textual descriptions passed to the LLM, as shown in figure 2.7.

Figure 2.7 Revisiting the tool-equipping process. With a BaseTool, we can use its attributes to fill in the values required for textual descriptions that are passed to the LLM.



Once a tool is equipped to the LLM, it can be used in a tool-call process. As we've already discussed, the `__call__()` method is now responsible for the tool's execution, which takes in a `ToolCall` and outputs a `ToolCallResult`.

We'll mark this `__call__()` method as abstract, implying that subclasses will need to provide an implementation for it. The following listing shows the code for `BaseTool`.

Listing 2.3 Implementing BaseTool

```
# llm_agents_from_scratch/base/tool.py  #A
from abc import ABC, abstractmethod
from llm_agents_from_scratch.data_structures.tool import (
    ToolCall,  #B
    ToolCallResult,  #B
)

class BaseTool(ABC):
    """Base Tool Class."""

    @property  #C
    @abstractmethod
    def name(self) -> str:
        """Name of tool."""


    @property
    @abstractmethod  #D
    def description(self) -> str:
        """Description of what this tool does."""

    @property
    @abstractmethod
    def parameters_json_schema(self) -> dict[str, Any]:
        """JSON Schema for tool parameters."""

    @abstractmethod
    def __call__(
        self,
        tool_call: ToolCall,
        *args: Any,
        **kwargs: Any,
    ) -> ToolCallResult:
        """Execute the tool call."""  #E
```

You can see that we've marked the `__call__()` method as abstract by applying the `@abstractmethod` decorator to it.

For our three attributes, you may be surprised to see that we have marked them with `@abstractmethod` as well. While they're marked as abstract methods, the `@property` decorator gives us the attribute-like behavior we want. This provides some flexibility in terms of hiding internal attributes and performing validation

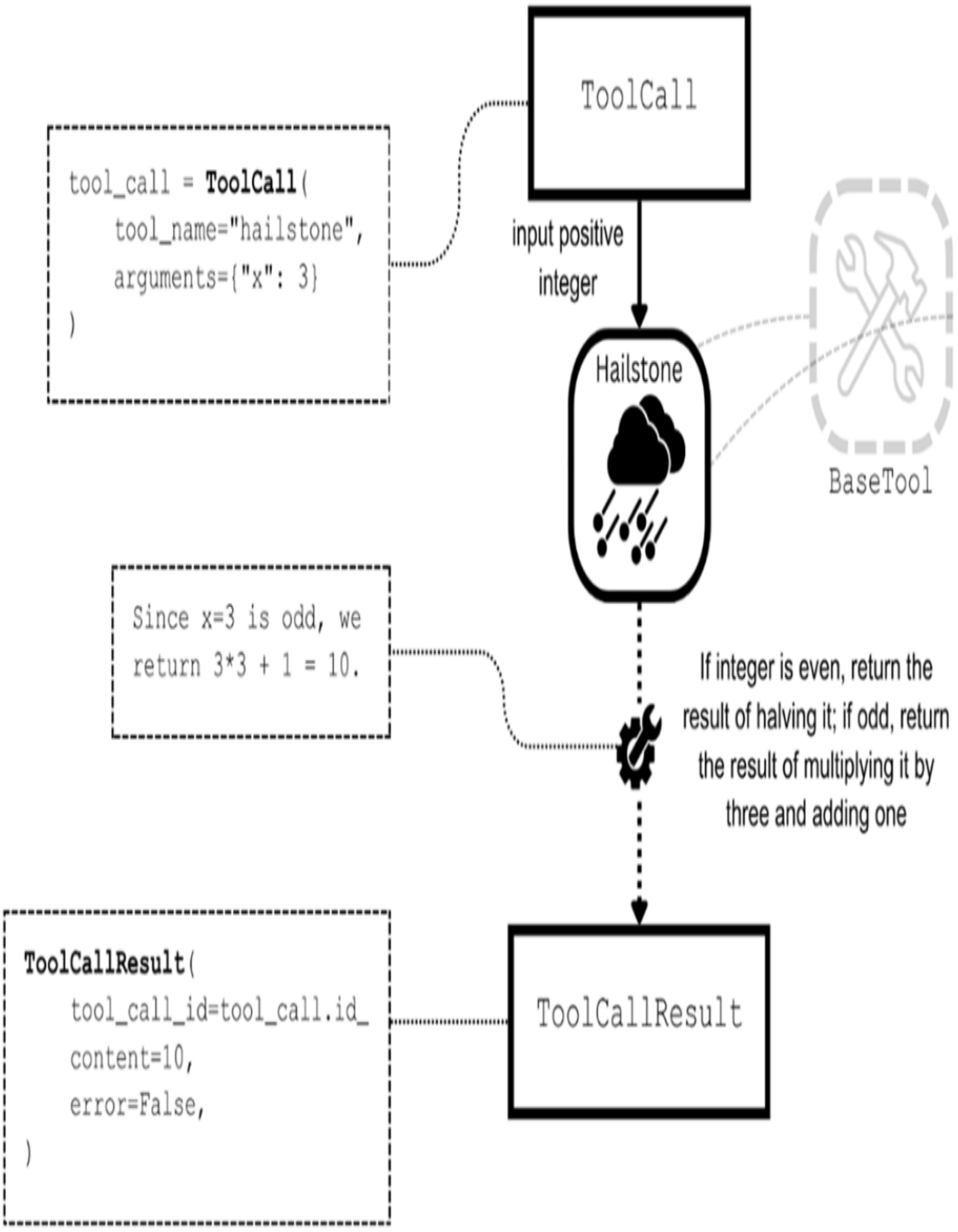
when needed. However, the main reason I have chosen to use property attributes here is for a consistent interface. All subclasses must provide implementations for `name`, `description`, `parameters_json_schema`, and `__call__()`, and failing to do so would raise an `AbstractMethodError`.

NOTe

Most, if not all, LLM providers have designed their LLM tool-calling APIs and SDKs to accept outlines of the tool's parameters that adhere to the JSON Schema specification. Using JSON Schema for our framework is a technically sound choice that also happens to maximize compatibility with existing LLM tools and services.

Let's look at a simple example of building a tool by subclassing the `BaseTool` class we just implemented. For this example, we'll create a tool called the Hailstone tool that applies a single step of the Hailstone sequence to a given positive integer. If the integer is even, the Hailstone tool outputs the result of halving it; if odd, the Hailstone tool outputs the result of multiplying it by three and adding one. Figure 2.8 illustrates the Hailstone tool and its execution logic as part of a tool-call process.

Figure 2.8 The Hailstone tool performs a Hailstone step on a given positive integer. By implementing the Hailstone tool as a subclass of BaseTool, we can use it within the tool-call process for LLMs.



To implement the Hailstone tool as a `BaseTool`, we know from before that we'll have to write implementations for `name`, `description`, `parameters_json_schema`, and `__call__()`.

For `name` and `description`, we'll use "hailstone" and "A tool that performs a Hailstone step on a given input number," respectively. For

`parameters_json_schema`, we need to provide a schema of the Hailstone tool's parameters that conforms to the JSON Schema specification. We'll create this manually for this simple example, but later in this chapter, we'll build a helper function to generate these JSON Schemas automatically.

The Hailstone tool requires only one input parameter: the integer to which we want to apply the Hailstone step. If we call this input parameter `x`, which aligns with the naming used in figure 2.8, the JSON Schema for the Hailstone tool's parameters looks like the following code.

```
{
    "type": "object",
    "properties": {  #A
        "x": {  #B
            "type": "number",
            "description": "The input number."
        },
    },
    "required": ["x"],  #C
}
```

As you can see, the input parameter `x` is specified and listed as required.

JSON Schemas like the one we just created are simply JSON documents that use a set of reserved keywords to describe their data requirements. Readers unfamiliar with the JSON Schema specification may find it helpful to read the sidebar, which provides an overview of the basics of JSON Schema.

JSON Schema basics

JSON Schemas are themselves JSON documents that are used to describe the shape and format of data. This specification makes it easier to create and share data. For example, JSON Schemas can be used to specify the shape of data representing transactions from a retail store. As another example, a JSON Schema can also be used to describe LLM agents.

Let's say that we want to have LLM agents specified by their backbone LLM and the list of its equipped tools. To accomplish this, we'd structure the overall JSON Schema using the type `object`. This `object` JSON data type is analogous to the Python `dict` type. Within an `object`, we can define its `properties`, or key-value pairs, that represent the object.

For our LLM agent, we would have two properties: `llm` and `tools`. Each property in JSON Schema is defined by a schema object that specifies its data type and constraints. We may want the `llm` property, for example, to specify the name of the LLM model, like "gpt-5". In this case, we'd specify the `llm` property type to be the `string` JSON type.

In contrast, the `tools` property would be an `array` JSON type, which is analogous to the Python `list` data type. In addition to specifying the data type, we can also provide a name or `title` to each JSON Schema fragment. The following code shows the JSON Schema that we just described.

```
{
    "title": "LLMAgent",
    "type": "object",  #A
    "properties": {
        "llm": {  #B
            "title": "Llm",
            "type": "string"  #C
        },
        "tools": {  #D
            "items": {  #E
                "additionalProperties": true,
                "type": "object"
            },
            "title": "Tools",
            "type": "array"  #F
        },
    },
    "required": [  #G
        "llm",
        "tools"
    ]
}
```

With this JSON Schema, we know precisely how to create a valid LLM agent data record. We see the shape and format of this data and understand that a valid LLM agent data record must include both `llm` and `tools` properties, as shown in the following code.

```
{
    "llm": "gpt-5",
    "tools": [
        {
            "name": "websearch-tool"
        }
    ]
}
```

Failing to have either of these fields or supplying incorrect data types would result in a JSON data validation error. The following code shows an invalid LLM agent data record because it is missing the `llm` property.

```
{
    "tools": [
        {
            "name": "websearch-tool"
        }
    ]
}
```

By passing JSON Schemas of the tool parameters to LLMs, they'll know exactly how to provide the required information for parameters when making tool-call requests.

For more comprehensive information on JSON Schemas, readers are encouraged to read other resources such as https://json-schema.org/.

The last item to implement for our Hailstone tool is its `__call__()` method. With our parameter's JSON Schema established, we can now expect `ToolCall` objects passed to the Hailstone tool to contain the key `x` in their `arguments` dictionary. The value for this key is the input number to which we'll apply the Hailstone step logic. The following code implements this logic after extracting the input number from a given `ToolCall` object called `tool_call`.

```
x = tool_call.arguments.get("x")  #A
if x % 2 == 0:
    result = x // 2  #B
else:
    result = (x * 3) + 1  #C

return ToolCallResult(  #D
    tool_call_id=tool_call.id_,
    content=result,
    error=False,
)
```

Our code successfully implements the Hailstone step and packages the result into a `ToolCallResult`, as required by the `__call__()` method. This implementation keeps things simple at this stage, but a more robust version could include validation and error handling for the input parameter `x`.

The following code ties everything back together, providing the entire implementation for the Hailstone tool.

```
# Included in examples/ch02.ipynb  #A
from typing import Any
from llm_agents_from_scratch.base.tool import BaseTool
from llm_agents_from_scratch.data_structures.tool import (
   ToolCall,
   ToolCallResult,
)


class Hailstone(BaseTool):  #B
    @property
    def name(self) -> str:
        return "hailstone"

    @property
    def description(self) -> str:
        return "A tool that performs a Hailstone step on a given
input
            number."

    @property
    def parameters_json_schema(self) -> dict[str, Any]:
        """JSON Schema for tool parameters."""
        return {
            "type": "object",
            "properties": {
                "x": {
                    "type": "number",
                    "description": "The input number."
                },
            },
            "required": ["x"]
        }

    def __call__(
        self,
        tool_call: ToolCall,
        *args: Any,
        **kwargs: Any,
    ) -> ToolCallResult:
        """Execute the tool call."""
        x = tool_call.arguments.get("x")
        if x % 2 == 0:
            result = x // 2
        else:
            result = (x * 3) + 1

        return ToolCallResult(
            tool_call_id=tool_call.id_,
            content=result,
            error=False,
        )
```

We can run the Hailstone tool call shown in figure 2.8 as follows:

```
# Included in examples/ch02.ipynb  #A
hailstone_tool = Hailstone()  #B

tool_call = ToolCall(  #C
    tool_name="hailstone",
    arguments={"x": 3},
)

tool_call_result = hailstone_tool(tool_call)  #D
print(tool_call_result)
```

The returned `ToolCallResult` object contains the result of applying the Hailstone step to `x = 3`, which is 10.

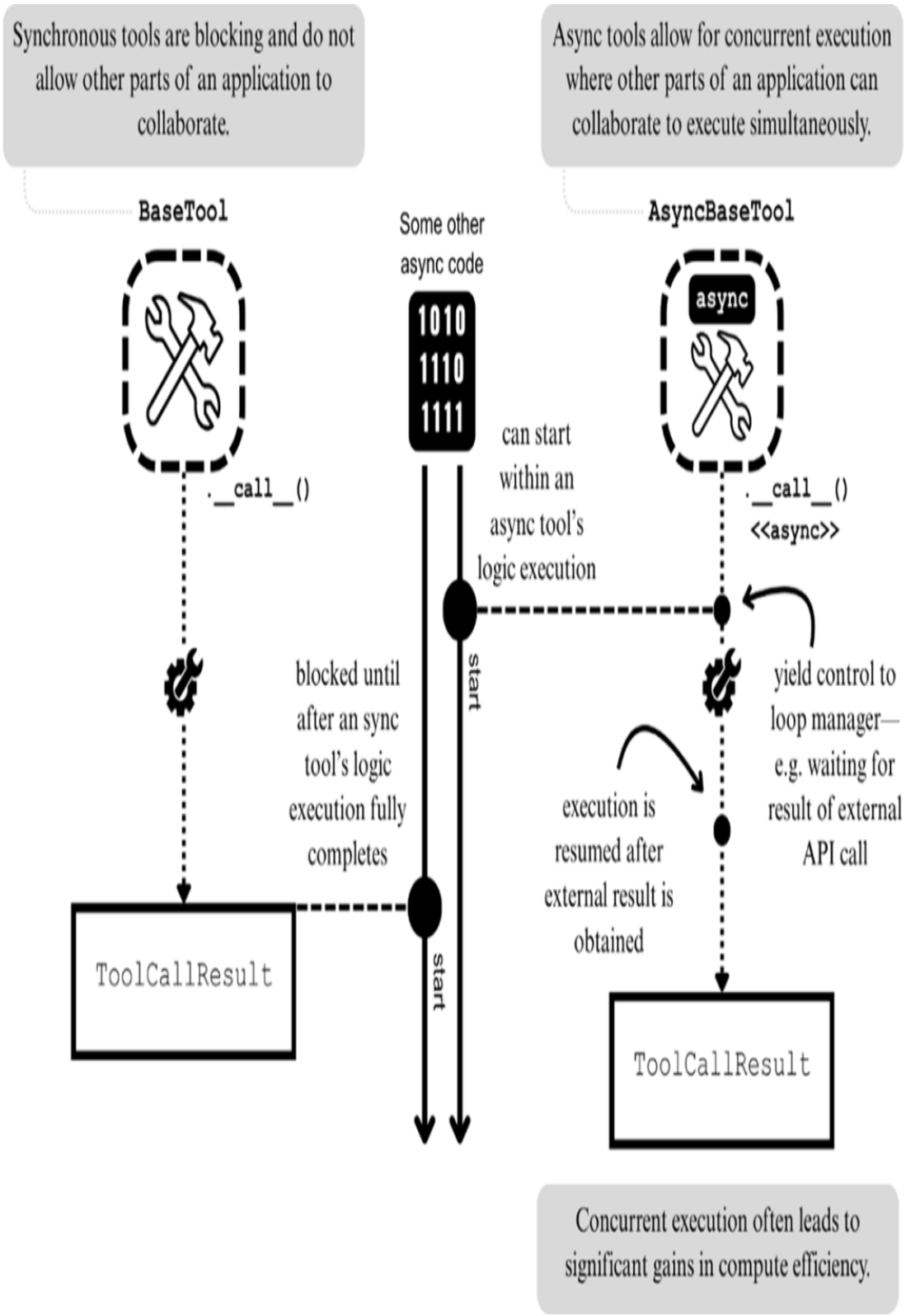
```
tool_call_id='112233', content='10', error=False
```

A demonstration of using the Hailstone tool's `name`, `description`, and `parameters_json_schema` to prepare a textual description for an LLM will have to wait until we've implemented the `BaseLLM` class, which is the subject of the next chapter. For now, let's go over the other base class that we mentioned earlier: `AsyncBaseTool`.

2.1.3 The AsyncBaseTool

To conclude this section, we'll present an important variation of the `BaseTool` class: the `AsyncBaseTool`, which is designed for tools whose logic executes asynchronously. Tools that make external API calls, such as for checking weather data, are best suited for asynchronous execution. While waiting for the result of an external API call, asynchronous tools allow other parts of an application's code to execute simultaneously. Figure 2.9 illustrates the difference in execution mechanics between `BaseTool`, which is synchronous and therefore blocking, and the non-blocking `AsyncBaseTool`.

Figure 2.9 Comparing synchronous and asynchronous tool executions.

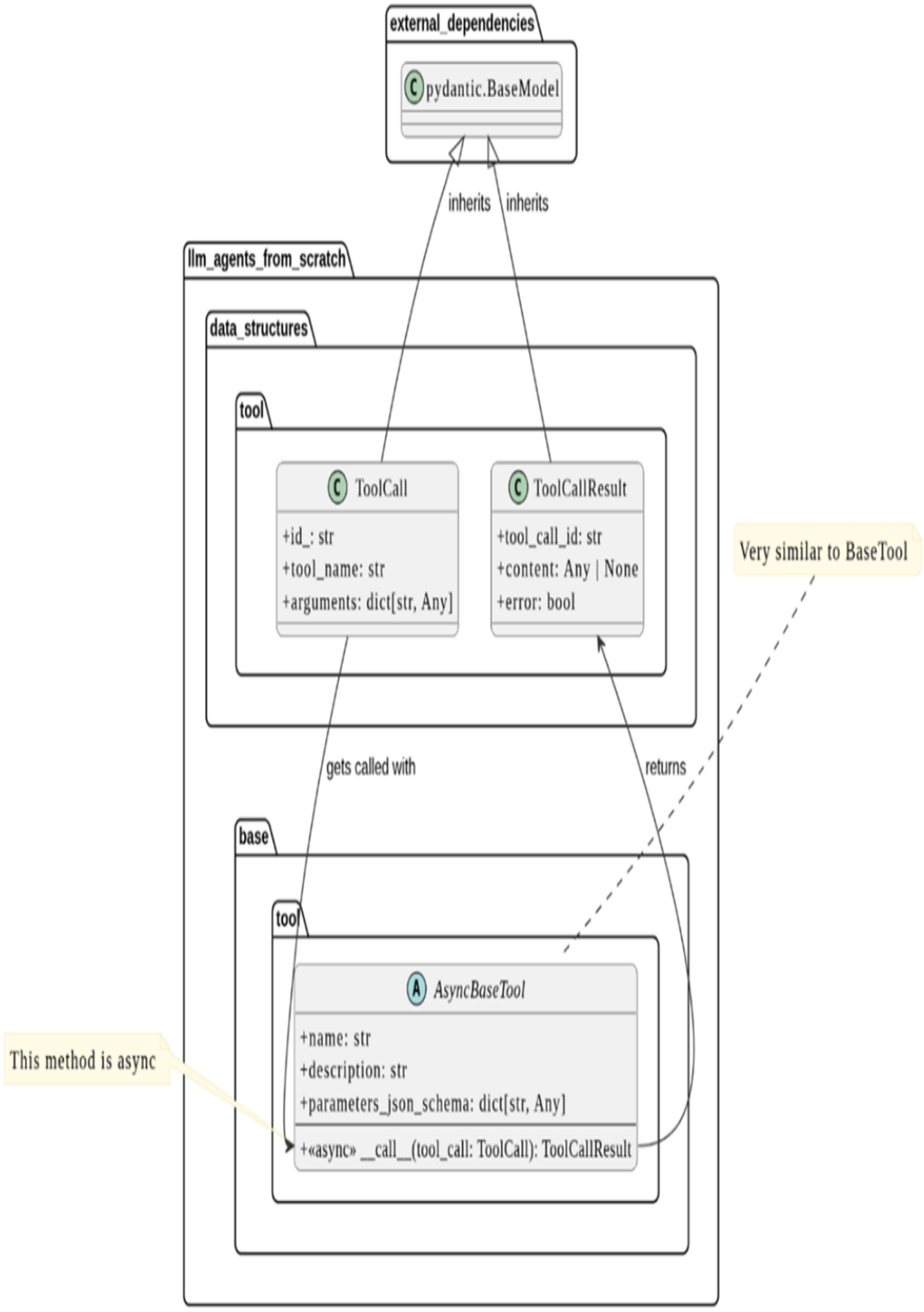


During the execution of an `AsyncBaseTool`, if it must wait for external results, other parts of an application can start or continue running. This concurrent

execution often yields significant speed gains due to improved resource utilization compared with the synchronous execution of its `BaseTool` counterpart.

Figure 2.10 shows the UML class diagram of the `AsyncBaseTool` class.

Figure 2.10 The UML class diagram for AsyncBaseTool, which shows a nearly identical structure to the BaseTool class.



The `AsyncBaseTool` class exposes an interface similar to `BaseTool`. It shares the same attributes as `BaseTool`, and also supports a `__call__()` method. However,

the `__call__()` method of `AsyncBaseTool` is asynchronous.

In Python, we mark methods as asynchronous by using the keyword `async` just before declaring the method signature. The complete implementation for `AsyncBaseTool` is provided in the following code.

Listing 2.4 Implementing AsyncBaseTool

```
# llm_agents_from_scratch/base/tool.py
from abc import ABC, abstractmethod
from llm_agents_from_scratch.data_structures.tool import (
    ToolCall,
    ToolCallResult,
)

class BaseTool(ABC):
    """Base Tool Class."""
    …  #A

class AsyncBaseTool(ABC):
    """Async Base Tool Class."""

    @property
    @abstractmethod
    def name(self) -> str:
        """Name of tool."""

    @property
    @abstractmethod
    def description(self) -> str:
        """Description of what this tool does."""

    @property
    @abstractmethod
    def parameters_json_schema(self) -> dict[str, Any]:
        """JSON schema for tool parameters."""

    @abstractmethod
    async def __call__(  #B
        self,
        tool_call: ToolCall,
        *args: Any,
        **kwargs: Any,
    ) -> ToolCallResult:
        """Asynchronously execute the tool call."""
```

As an example, if the Hailstone tool from earlier inherited from `AsyncBaseTool` instead of `BaseTool`, we would invoke it with the following code.

```
hailstone_tool = Hailstone()  #A

tool_call = ToolCall(  #B
    tool_name="hailstone",
    arguments={"x": 3},
)

tool_call_result = await hailstone_tool(tool_call)  #C
```

note

The `async` keyword turns a function into an asynchronous one that returns a coroutine when called. Coroutines are non-blocking objects that enable concurrent execution within an asynchronous event loop, such as the ones provided by the `asyncio` library. Results of coroutines must be awaited using the `await` keyword when called from an asynchronous method. To run a coroutine from synchronous code, you can use `asyncio.run()`.

Exercise 2.1 Hailstone as an AsyncBaseTool

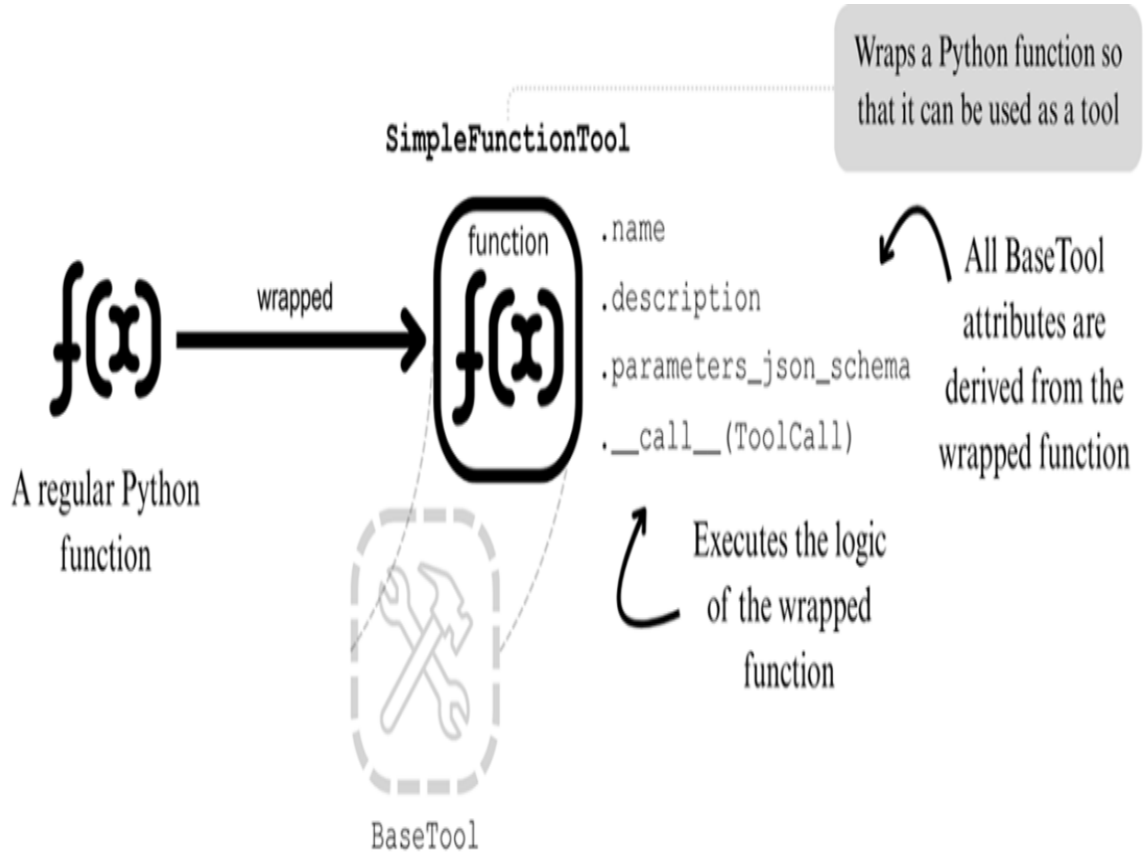
Re-implement the Hailstone tool from earlier, but this time make it inherit from `AsyncBaseTool`. For its execution logic, introduce a 1-second sleep before performing the Hailstone step by using `asyncio.sleep(1)`.

2.2 SimpleFunctionTool: a subclass of BaseTool

While we were able to implement the `Hailstone` tool with relative ease, we can add extra convenience by developing an abstraction that automatically builds `BaseTool` objects from Python functions. In this section, we'll create `SimpleFunctionTool`, a subclass of `BaseTool` that serves as a wrapper class for creating tools in this manner.

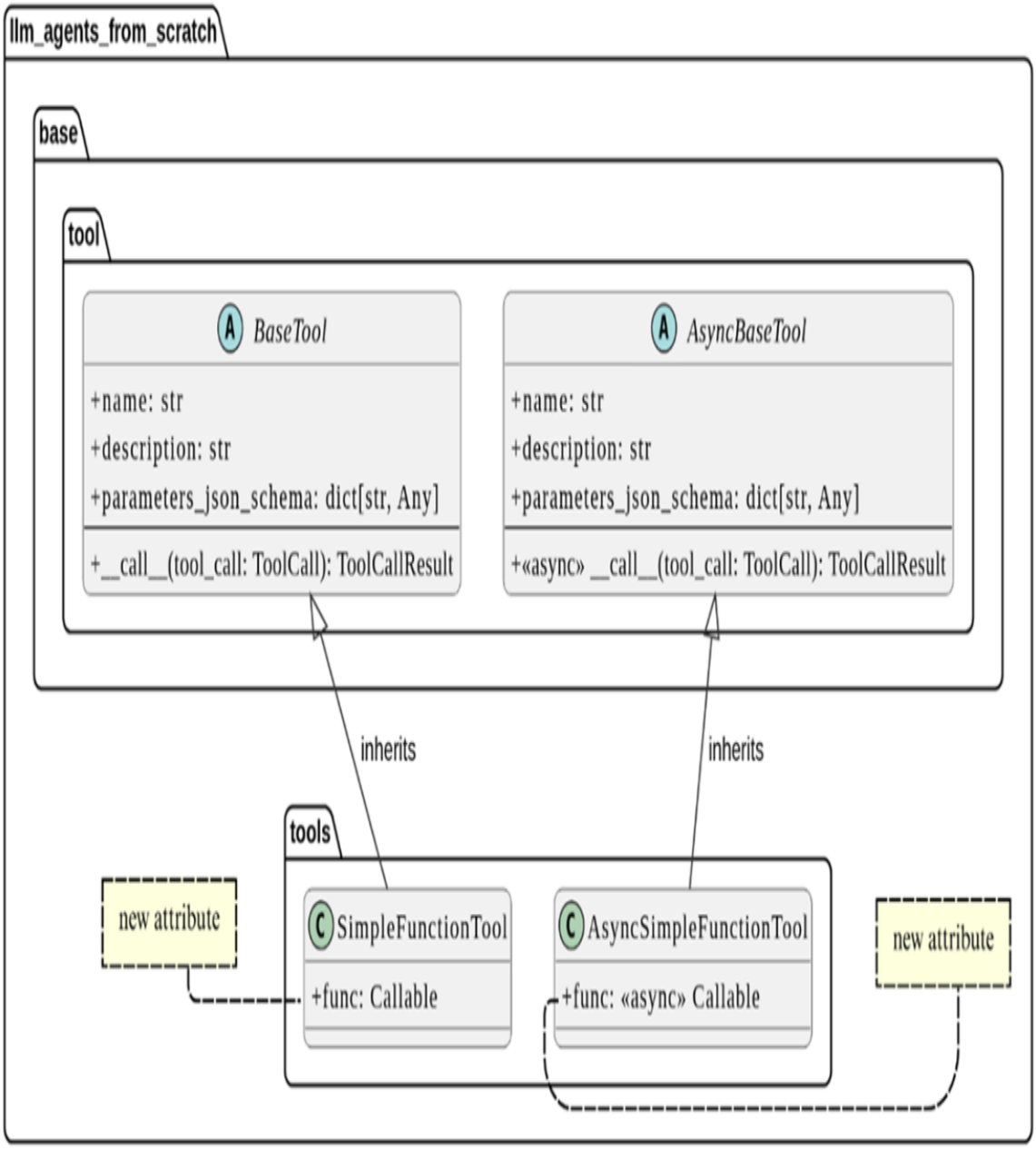
When supplied with a function, `SimpleFunctionTool` automatically implements `name`, `description`, `parameters_json_schema`, and `__call__()` using the information encapsulated in the function, as shown in Figure 2.11.

Figure 2.11 Wrapping Python functions with SimpleFunctionTool so that they can be used as tools for LLMs and LLM agents.



We'll also implement the asynchronous version of `SimpleFunctionTool`, for creating `AsyncBaseTool` objects derived from asynchronous Python functions. Figure 2.12 shows the UML class diagrams for `SimpleFunctionTool` and `AsyncSimpleFunctionTool`.

Figure 2.12 UML class diagrams for SimpleFunctionTool and AsyncSimpleFunctionTool.



Both classes inherit from their corresponding base tool classes and add a new attribute called `func`, which is the function they wrap. For `SimpleFunctionTool`, the wrapped function is synchronous, while for `AsyncSimpleFunctionTool`, it is asynchronous.

If we reconsider the Hailstone tool, the new `SimpleFunctionTool` class enables an alternative implementation based on a function that implements the Hailstone step logic, as shown in the following code.

```
# Included in examples/ch02.ipynb  #A
def hailstone_step_func(x: int) -> int:
    """Performs a single step of the Hailstone sequence."""
    if x % 2 == 0:
        return x // 2  #B
    else:
        return 3 * x + 1  #B
```

The idea is to have `SimpleFunctionTool` wrap the `hailstone_step_func()` to create a new tool that an LLM or LLM agent can use in a tool-call process.

2.2.1 Implementing SimpleFunctionTool

Now that we have an idea of what we're trying to accomplish, let's implement `SimpleFunctionTool`. Since `SimpleFunctionTool` inherits from `BaseTool`, as shown in figure 2.12, we know that we'll need to implement `name`, `description`, `parameters_json_schema`, and `__call__()`. As previously discussed, we'll derive these implementations automatically based on the supplied function and a new attribute, `func`.

To start, for the `name` attribute, we'll use the name of the Python function. For `description`, we can use the docstring of the Python function if it exists; otherwise, we fall back to some pre-specified template. The implementations of these first two attributes, as well as the constructor `__init__()` method are shown in listing 2.5.

Listing 2.5 Implementing SimpleFunctionTool.(__init__(), name, description)

```
# llm_agents_from_scratch/tools/simple_function.py
from typing import Any, Callable
from llm_agents_from_scratch.base.tool import BaseTool


class SimpleFunctionTool(BaseTool):
    """Simple function calling tool.

    Turn a Python function into a tool for an LLM.
    """
    def __init__(
        self,
        func: Callable[..., Any],  #A
        desc: str | None = None,
    ) -> None:
        """Initialize a SimpleFunctionTool.

        Args:
```

```
            func (Callable): The Python function to expose as a tool
to the
                LLM.
            desc (str | None, optional): Description of the function.
                Defaults to None.
        """
        self.func = func
        self._desc = desc  #B

    @property
    def name(self) -> str:
        """Name of function tool."""
        return self.func.__name__  #C

    @property
    def description(self) -> str:
        """Description of what this function tool does."""
        return (
            self._desc or self.func.__doc__ or f"Tool for
                {self.func.__name__}"  #D
        )
```

Let's now move on to implementing `parameters_json_schema` for `SimpleFunctionTool`. When we implemented the Hailstone tool earlier, we created the parameters JSON Schema manually. I mentioned then that we'd later build a helper function to automate the generation of these schemas. We'll do this now by implementing a helper that derives the JSON Schema from `func` by inspecting its method signature. We'll call this helper function: `function_signature_to_json_schema()`.

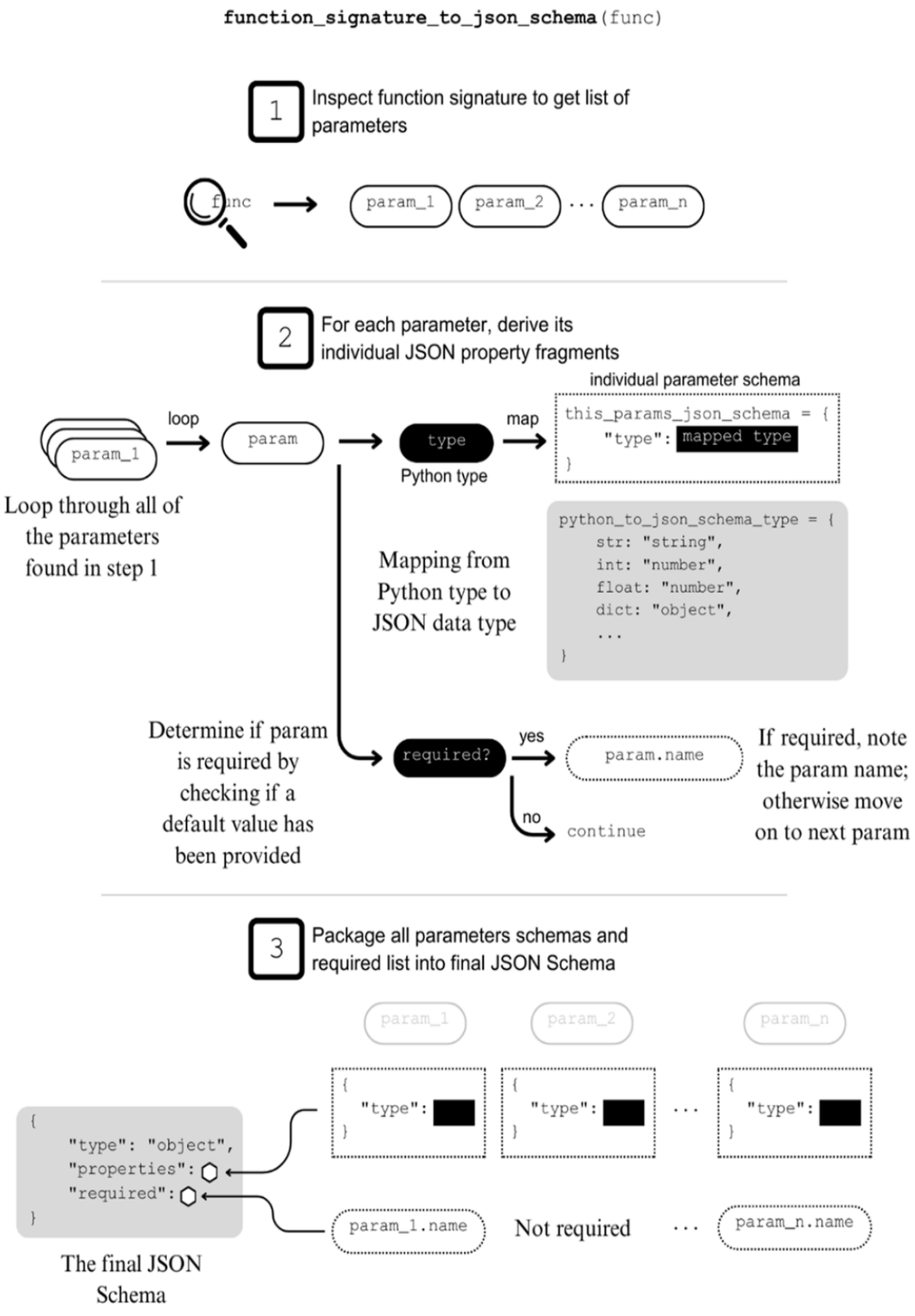
Implementing `function_signature_to_json_schema()` mainly involves mapping Python data types to the corresponding JSON Schema data types, as well as determining which of the function parameters are required because they have no defined default values.

NOTE

We're coding a JSON Schema builder from scratch with our implementation of `function_signature_to_json_schema()` to deepen our understanding of how the textual descriptions of tools are prepared for an LLM to consume. In the next section, we will introduce yet another `BaseTool` subclass that handles this aspect more robustly by leveraging the JSON Schema generation capabilities offered by the `pydantic` library.

There is quite a bit of logic that needs to be implemented in `function_signature_to_json_schema()`. To help you better understand all of it, I've illustrated the logic in figure 2.13. Let's walk through all of it together before seeing the entire implementation in a listing.

Figure 2.13 A visual breakdown of the three steps involved in the implementation of the helper function_signature_to_json_schema.



As shown in figure 2.13, there are three steps to implement. First, we introspect the function signature to identify the parameters and their annotated types. Second,

we loop through all the identified parameters to build a JSON Schema fragment for each of them. This step involves mapping the parameter's type to the corresponding JSON data type and determining if the parameter is required. Third, we assemble and return the overall JSON Schema.

The complete implementation of `function_signature_to_json_schema()` is provided next in listing 2.6.

Listing 2.6 Helper for turning a function signature into a JSON Schema

```
# llm_agents_from_scratch/tools/simple_function.py
import inspect
from typing import Any, Callable, get_type_hints

def function_signature_to_json_schema(func: Callable) -> dict[str,
Any]:
    """Turn a function signature into a JSON schema.

    Inspects the signature of the function and maps types to the
    appropriate JSON schema type.

    Args:
        func (Callable): The function for which to get the JSON
schema.

    Returns:
        dict[str, Any]: The JSON schema
    """
    sig = inspect.signature(func)  #A
    type_hints = get_type_hints(func)
    python_to_json_schema_type = {  #B
        str: "string",
        int: "number",
        float: "number",
        dict: "object",
        list: "array",
        type(None): "null",
        bool: "boolean",
        tuple: "array",
        bytes: "string",
        set: "array",
    }

    properties = {}
    required = []
    for param in sig.parameters.values():  #C
        # skip args and kwargs
        if param.kind in (param.VAR_POSITIONAL, param.VAR_KEYWORD):
            continue
```

```
        annotation = type_hints.get(param.name, param.annotation)
        if annotation in python_to_json_schema_type:
            this_params_json_schema = {
                "type": python_to_json_schema_type[annotation],
            }
        else:
            # fallback schema, that accepts everything
            this_params_json_schema = {}
        properties[param.name] = this_params_json_schema

        # check if param is required
        if param.default == inspect._empty:  #D
            required.append(param.name)

    return {  #E
        "type": "object",
        "properties": properties,
        "required": required,
    }
```

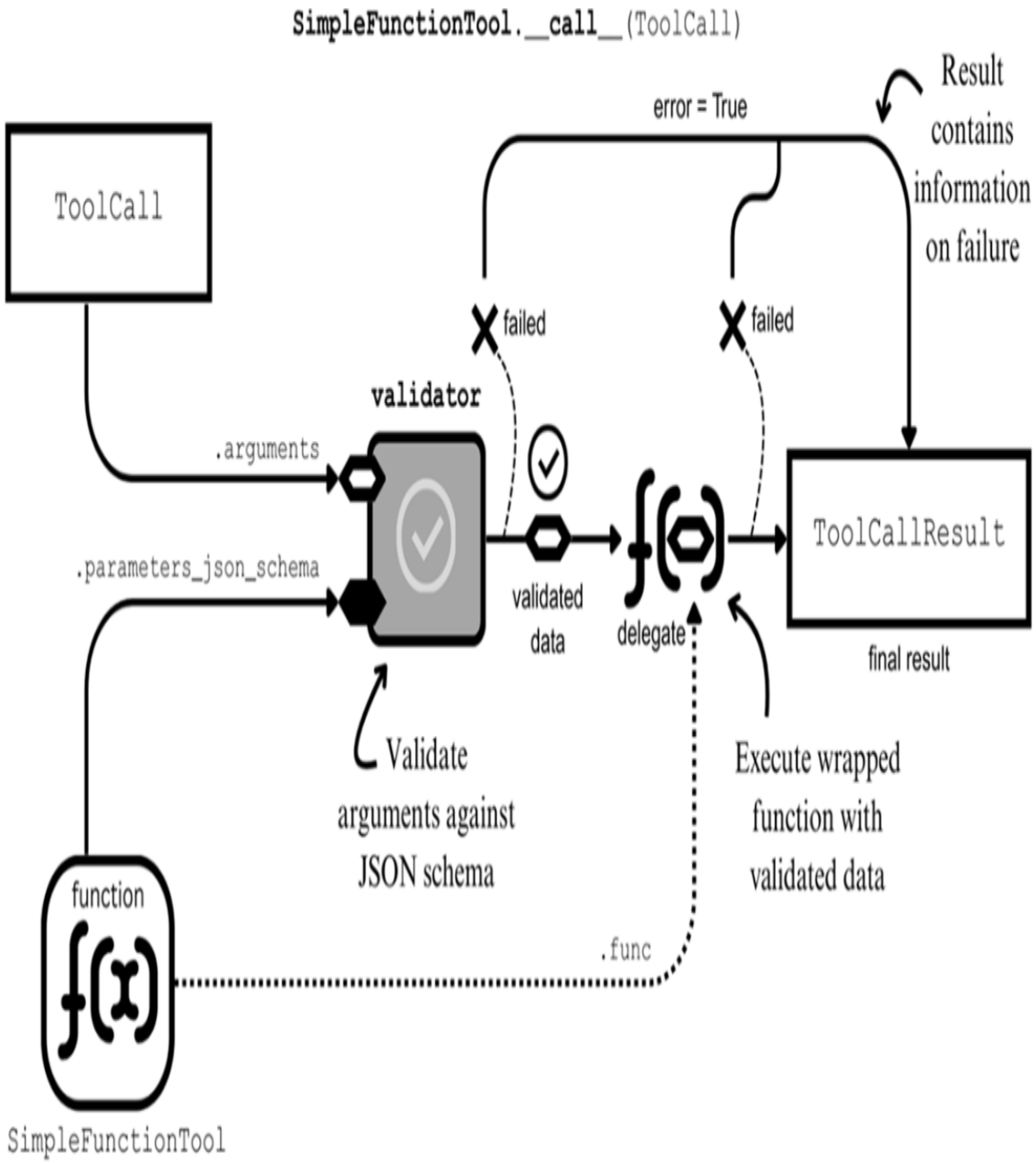
Phew, that was a lot of work. Fortunately, all that remains to implement `parameters_json_schema` is to invoke our helper function, as shown in the following code.

Listing 2.7 Implementing SimpleFunctionTool.`parameters_json_schema`

```
# llm_agents_from_scratch/tools/simple_function.py
import inspect
from typing import Any, Callable, get_type_hints
from llm_agents_from_scratch.base.tool import BaseTool


def function_signature_to_json_schema(func: Callable) -> dict[str,
Any]:
    …  #A

class SimpleFunctionTool(BaseTool):
    """Simple function calling tool.

    Turn a Python function into a tool for an LLM.
    """
    …  #B

    @property
    def parameters_json_schema(self) -> dict[str, Any]:
        """JSON schema for tool parameters."""
        return function_signature_to_json_schema(self.func)  #C
```

Let's move on to the final required implementation, `__call__()`. Since `SimpleFunctionTool` serves as a wrapper class to create a tool from `func`, we'll simply delegate to `func` within `__call__()`. Before this delegation, however, we'll first perform validation on the parameter data provided by the LLM in its tool-call request. Figure 2.14 shows these validation and delegation steps when `__call__()` is invoked.

Figure 2.14 Executing a SimpleFunctionTool involves validating the parameter data of the ToolCall object and subsequently delegating to the wrapped function.



Any failures experienced in either the validation or delegation stages are handled by returning a `ToolCallResult` with `error` set to `True` and `content` set to a JSON-serialized string containing information on the error.

To perform the parameter data validation, we'll use the `jsonschema` library. When a `ToolCall` object, say `tool_call`, is passed to `__call__()`, we validate `tool_call.arguments` against the tool's `parameters_json_schema`. The following listing shows this validation aspect of the overall `__call__()` implementation.

Listing 2.8 Implementing SimpleFunctionTool.__call__() (validation)

```
# llm_agents_from_scratch/tools/simple_function.py
…  #A
from jsonschema import SchemaError, ValidationError, validate

class SimpleFunctionTool(BaseTool):
    """Simple function calling tool.

    Turn a Python function into a tool for an LLM.
    """
    …  #B

    def __call__(
        self,
        tool_call: ToolCall,
        *args: Any,
        **kwargs: Any,
    ) -> ToolCallResult:
        …  #C
        try:
            # validate the arguments
            validate(tool_call.arguments,  #D
                   schema=self.parameters_json_schema)
        except (SchemaError, ValidationError) as e:
            error_details = {
                "error_type": e.__class__.__name__,
                "message": e.message,
            }
            return ToolCallResult(  #E
                tool_call_id=tool_call.id_,
                content=json.dumps(error_details),
                error=True,
            )

        …  #F
```

Delegating to the wrapped function, `func`, amounts to passing the validated `tool_call.arguments` to it. Listing 2.9 implements the delegation portion of

`__call__()`.

Listing 2.9 Implementing SimpleFunctionTool.__call__() (delegation)

```
# llm_agents_from_scratch/tools/simple_function.py
…  #A


class SimpleFunctionTool(BaseTool):
    """Simple function calling tool.

    Turn a Python function into a tool for an LLM.
    """
    …  #B

    def __call__(
        self,
        tool_call: ToolCall,
        *args: Any,
        **kwargs: Any,
    ) -> ToolCallResult:
        …  #C

        …  #D

        try:
            # execute the function
            res = self.func(**tool_call.arguments)  #E
        except Exception as e:
            error_details = {
                "error_type": e.__class__.__name__,
                "message": f"Internal error while executing tool:
                    {str(e)}",
            }
            return ToolCallResult(  #F
                tool_call_id=tool_call.id_,
                content=json.dumps(error_details),
                error=True,
            )

        return ToolCallResult(
            tool_call_id=tool_call.id_,
            content=str(res),
            error=False,
        )
```

We have now completed our implementation of `SimpleFunctionTool`. To celebrate, let's complete the alternative implementation of the Hailstone tool, based on the `hailstone_step_func()` we coded earlier.

```
# Included in examples/ch02.ipynb  #A
from llm_agents_from_scratch.tools.simple_function import (
    SimpleFunctionTool
)

# convert our Python function to a BaseTool
hailstone_tool = SimpleFunctionTool(hailstone_step_func)

print(hailstone_tool.name)
print(hailstone_tool.description)
print(hailstone_tool.parameters_json_schema)
```

The resulting print statements should return the values of the attributes that were automatically derived from the supplied `hailstone_step_func()`.

```
hailstone_step_func  #A
Performs a single step of the Hailstone sequence.  #B
{'type': 'object', 'properties': {'x': {'type': 'number'}},
'required': ['x']}  #C
```

Running this version of `hailstone_tool` with a `ToolCall` object works in the same manner as our original implementation.

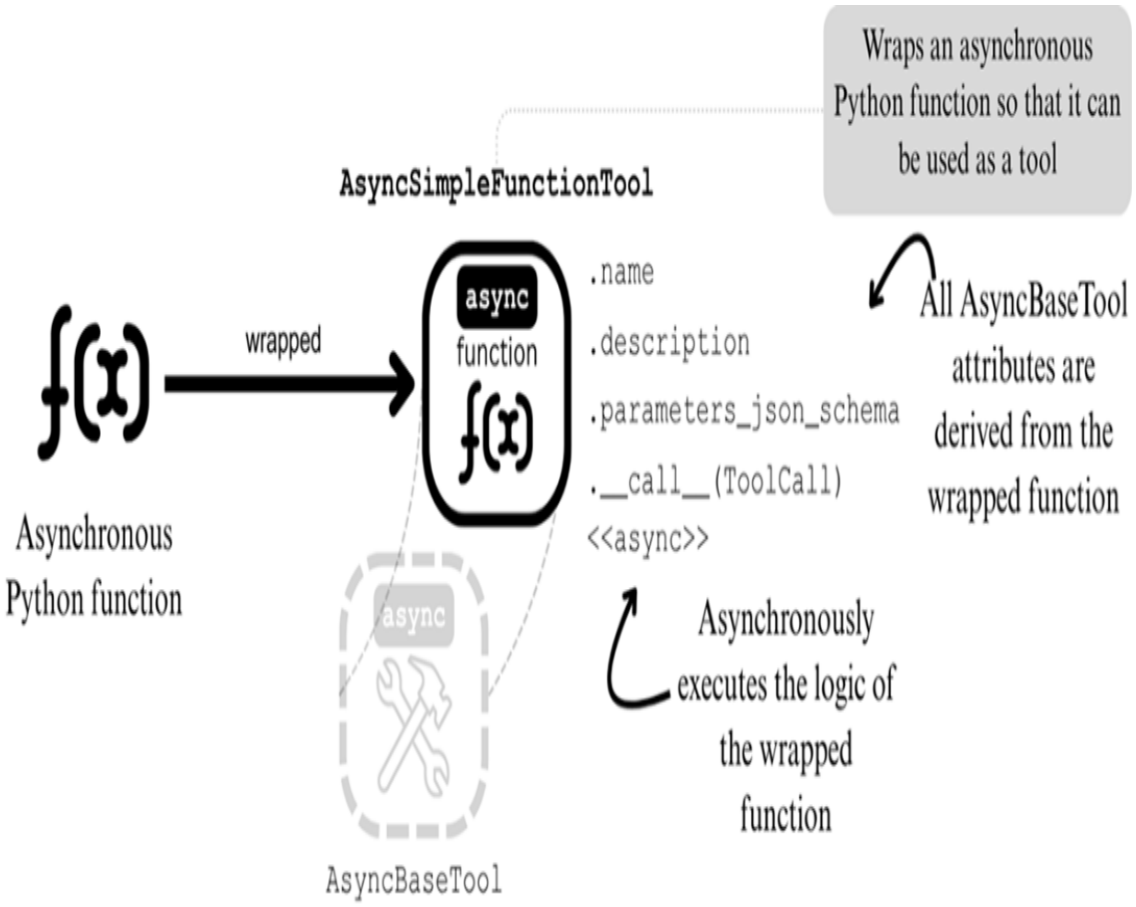
```
# Included in examples/ch02.ipynb  #A
from llm_agents_from_scratch.data_structures import ToolCall

tool_call = ToolCall(
    tool_name="hailstone_fn",
    arguments={"x": 3}
)

res = hailstone_tool(tool_call)  #B
```

This is great. With `SimpleFunctionTool`, we've unlocked a useful pattern for building tools from Python functions that LLMs and LLM agents can use. Let's wrap up this section by quickly implementing its asynchronous counterpart.

2.2.2 The AsyncSimpleFunctionTool

With `AsyncSimpleFunctionTool`, we aim to provide the same automatic tool creation that `SimpleFunctionTool` enabled, but for asynchronous functions. The mental model for `AsyncSimpleFunctionTool` is similar to that for `SimpleFunctionTool`, which was illustrated in figure 2.11. The adapted version for `AsyncSimpleFunctionTool` is shown in figure 2.15.

Figure 2.15 Wrapping an asynchronous Python function to create an AsyncBaseTool object automatically.



You might also recall from the UML class diagrams shown in figure 2.12 that `AsyncSimpleFunctionTool` and `SimpleFunctionTool` are structurally very similar. The main difference is that the `func` attribute of `AsyncSimpleFunctionTool` stores an asynchronous Python function.

All these shared similarities mean that we can reuse much of the code we wrote for `SimpleFunctionTool` when implementing `AsyncSimpleFunctionTool`. Because of this, we'll only cover the parts of the implementation where differences do exist between the two classes.

The first subtle difference lies in the implementation of the constructor, `__init__()`. Specifically, the type annotation that we'll use for `func` is more specialized now: `Callable[…, Awaitable[Any]]`. This updated annotation indicates that we're now working with a function that returns an object that can be awaited by using the `await` keyword. Coroutines, which we previously discussed, are one example of an awaitable object.

TIP

The `typing.Awaitable` class is more flexible than `typing.Coroutine`, representing any object that is awaitable, which includes coroutines, `asyncio.Futures`, and `asyncio.Tasks`.

The next and final difference involves the implementation of `__call__()`. Since the wrapped function is asynchronous, we'll need to use the `await` keyword when delegating to it. The validation logic is identical to that used for `SimpleFunctionTool`. The following code shows the implementation of `AsyncSimpleFunctionTool`.

Listing 2.10 Implementing AsyncSimpleFunctionTool (differences only)

```
# llm_agents_from_scratch/tools/simple_function.py
…  #A
from llm_agents_from_scratch.base.tool import AsyncBaseTool


class AsyncSimpleFunctionTool(AsyncBaseTool):  #B
    """Async simple function calling tool.

    Turn a Python function into a tool for an LLM.
    """

    def __init__(
        self,
        func: Callable[..., Awaitable[Any]],  #C
        desc: str | None = None,
    ) -> None:
        …  #D

    …  #E

    async def __call__(  #F
        self,
        tool_call: ToolCall,
        *args: Any,
        **kwargs: Any,
    ) -> ToolCallResult:
        …  #G

        …  #H

        try:
            # execute the function
            res = await self.func(**tool_call.arguments)  #I
        …  #J
```

Exercise 2.2 Alternative async Hailstone implementation

Re-implement the async Hailstone you created in Exercise 2.1, but this time using `AsyncSimpleFunctionTool`. To do this, you'll need to turn the `hailstone_step_func()` into an asynchronous function. Test both versions on the same input to verify they return identical outputs.

2.3 PydanticFunctionTool: another subclass of BaseTool

The `SimpleFunctionTool` from the previous section features our from-scratch helper method `function_signature_to_json_schema()` for automatically deriving JSON Schemas from function signatures. In this final section, we'll provide an alternative function tool wrapper class, `PydanticFunctionTool`, with similar capabilities to `SimpleFunctionTool`, but which leverages the `pydantic` library for more powerful and robust JSON Schema generation and validation capabilities.

The implementation procedure for `PydanticFunctionTool`, which also inherits from `BaseTool`, is very similar to that used to implement `SimpleFunctionTool` in the previous section. For this reason, we won't cover the full implementation of `PydanticFunctionTool` here and instead will focus on its usage pattern in our framework. Interested readers can refer to Appendix C for a comprehensive walkthrough of the full implementation.

The usage pattern for our `PydanticFunctionTool` is slightly different than that of `SimpleFunctionTool`. The main difference is that we'll now require the parameters of the wrapped function to be supplied via a `pydantic.BaseModel` as shown in the following code.

```
# Included in examples/ch02.ipynb  #A
from pydantic import BaseModel

class MyFuncParams(BaseModel):  #B
    x: int


def my_func(params: MyFuncParams) -> int:  #C
    print(params.x)  #D
```

With `my_func()` defined, we can use `PydanticFunctionTool` in a similar manner to `SimpleFunctionTool` to wrap `my_func()` to automatically create a tool that an

LLM or LLM agent can use.

```
# Included in examples/ch02.ipynb  #A
from llm_agents_from_scratch.tools.pydantic_function import (
    PydanticFunctionTool
)

tool = PydanticFunctionTool(my_func)  #B
```

This new `tool` can take in `ToolCall` objects to perform the tool-call process like any other tools implemented in this chapter.

Exercise 2.3 Hailstone tool as an PydanticFunctionTool

Re-implement the Hailstone tool, but this time using `PydanticFunctionTool`. Refer to Appendix C for usage guidance to complete this exercise.

The main benefits of `PydanticFunctionTool` are not directly obvious from the usage pattern we've just covered. Instead, the benefits are seen in the implementation of the `parameters_json_schema` attribute and the validation portion within `__call__()`. For these, we now rely on Pydantic's more robust JSON Schema generation and validation capabilities through `BaseModel.model_json_schema()` and `BaseModel.model_validate()`, respectively. In addition to the `PydanticFunctionTool`, its asynchronous counterpart, `AsyncPydanticFunctionTool` has also been added to the framework. Both classes can be imported from `llm_agents_from_scratch.tools.pydantic_function`.

We have covered a lot of ground in this chapter by implementing the base tool interfaces as well as adding a couple of handy tool factory classes that turn Python functions into tools which LLMs and LLM agents can use.

In the next chapter, we'll crucially implement our `BaseLLM` class as well as the `OllamaLLM` subclass of it that will allow us to work with LLMs supported by Ollama, a popular open-source LLM inference framework.

2.4 Summary

- To build an LLM agent, we need to build the required infrastructure, including abstractions representing tools that they work with to perform tasks.
- LLMs require the following information to be able to make a tool call request for a given tool: its name, a description of its functionality, and a JSON

Schema of its input parameters.

- A `BaseTool` object executes a single `ToolCall` and returns a single `ToolCallResult` object.
- The `AsyncBaseTool` class is designed for tools that execute their logic asynchronously.
- The `SimpleFunctionTool` is a wrapper class for turning Python functions into a `BaseTool` objects.
- The `AsyncSimpleFunctionTool` is a wrapper class for turning async Python functions into a `AsyncBaseTool` objects.
- The `PydanticFunctionTool` is similar to the `SimpleFunctionTool`, but wraps a special function we called `PydanticFunction` instead. These functions get passed their input parameters for logic execution through a `~pydantic.BaseModel`.

3 Working with LLMs

This chapter covers

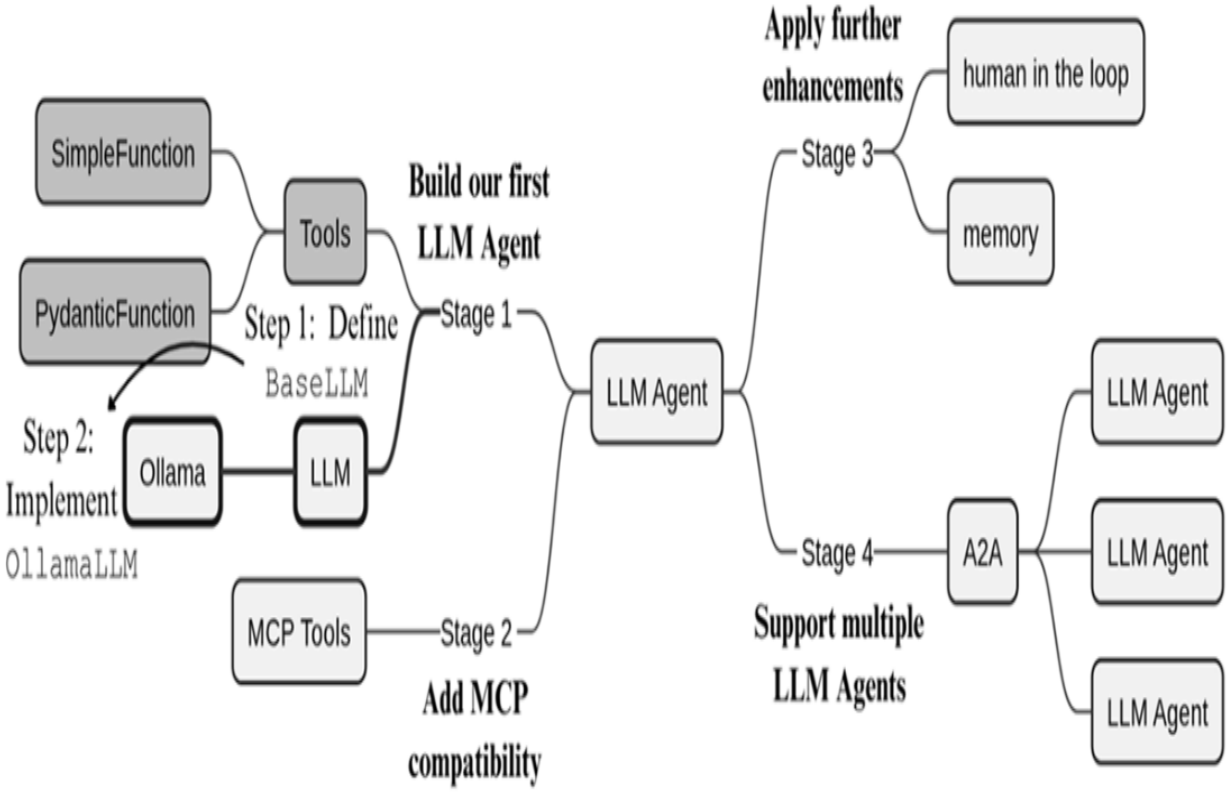
- The base class for working with LLMs in our LLM agent framework
- Implementatingr an LLM class that enables the use of any open-source LLM with Ollama
- A complete demonstration of the tool-call process

In the previous chapter, we began our Stage 1 build of `llm-agents-from-scratch` by writing base classes for tools as well as the necessary data structures that they work with. We'll continue our Stage 1 build here by similarly adding a base class for LLMs and the data structures that will enable the various modes of interacting with LLMs we want to support in our framework.

One such mode is the tool-calling process, which we'll finally be able to execute in its entirety by the end of this chapter. Specifically, we'll learn how to elicit a tool-call request from an LLM and how to submit the result of the tool invocation we covered in the previous chapter back to the LLM for synthesis and response.

After establishing our base class, `BaseLLM`, we'll move on to the very exciting task of building an integration with Ollama, a highly popular open-source LLM inference framework. We'll do this by implementing `OllamaLLM`, a subclass of `BaseLLM`, which will enable the use of any of the many open-source LLMs supported by Ollama, including those from the Llama and Qwen families of models. Figure 3.1 shows our updated build plan, highlighting the progress we've made so far and our current focus.

Figure 3.1 Having added tools to our LLM agent framework, the focus of this chapter is to add the other main component of LLM agents—their backbone LLM—to our framework. We'll specify the interface that all future LLMs must conform to through the BaseLLM class and implement the OllamaLLM subclass to enable the use of any of the open-source LLMs supported by Ollama.



As a reminder, you can follow along with the code examples by forking the book's GitHub repository and activating the framework's dedicated virtual environment as discussed in the previous two chapters. For added convenience, I've also prepared a Jupyter notebook to provide an execution environment for the coded examples in this chapter: https://github.com/nerdai/llm-agents-from-scratch/blob/main/examples/ch03.ipynb. Code snippets marked like the example code below are available in this notebook.

```
# Included in examples/ch03.ipynb  #A
…  #B
```

I recommend using `uv` to launch Jupyter Lab with all the necessary packages. Run the following terminal command from the project's root directory.

```
uv run --with jupyter jupyter lab
```

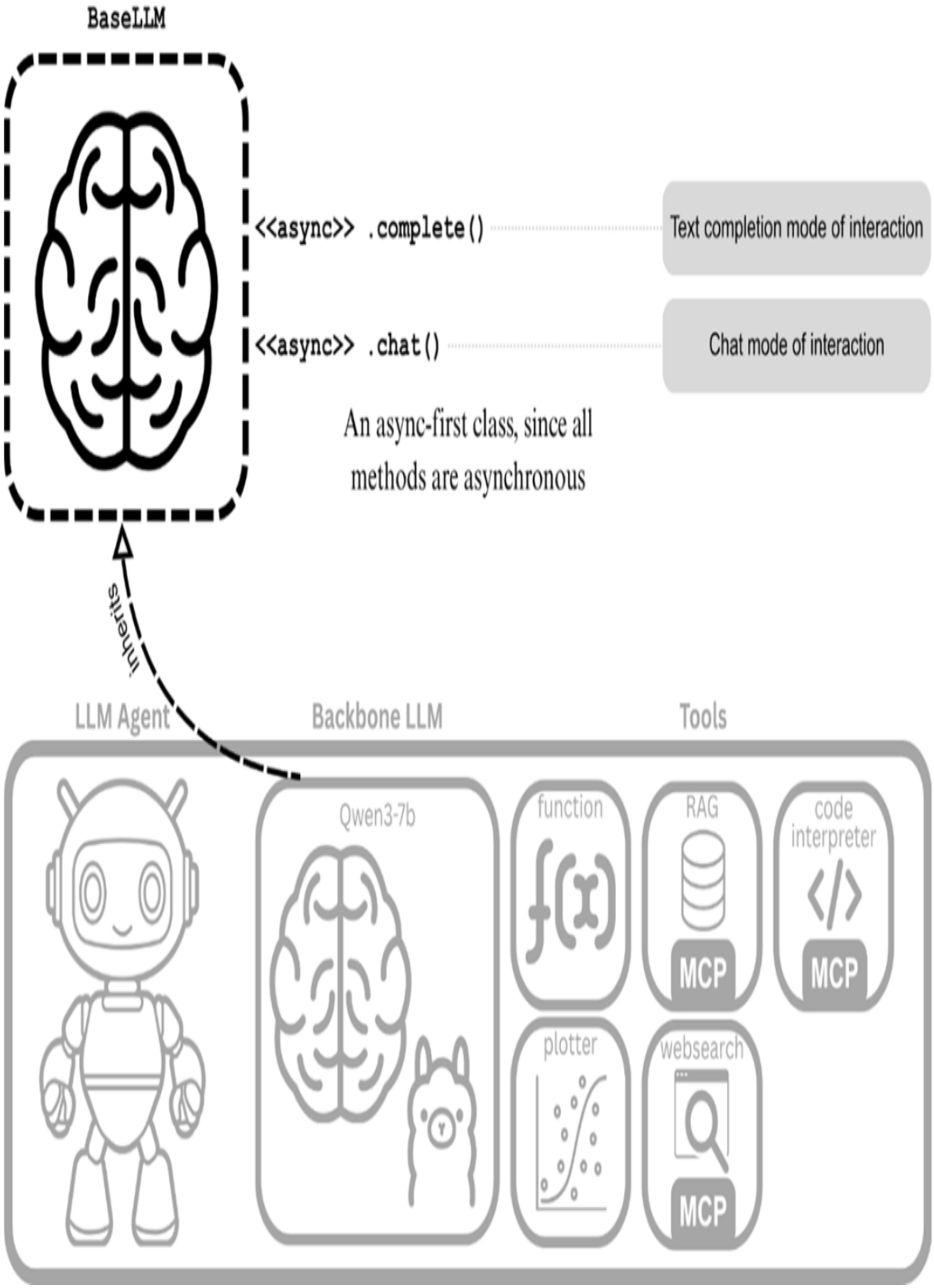
3.1 BaseLLM: a blueprint for LLMs

Several LLM providers exist today. OpenAI, with its GPT series, and Anthropic, with Claude, are two of the mainstream closed-source options. On the other hand, open-source LLMs, including those from the Llama, Qwen, and DeepSeek families of models, can be utilized through frameworks such as HuggingFace, Ollama, and vLLM. Interacting with LLMs from these providers and frameworks involves working with their respective APIs or SDKs. While all of them support the standard modes of interacting with an LLM, which we'll cover shortly, there are differences in how they can be used to build applications.

If we were to expose each of these APIs in our framework, it would become challenging and frustrating for us and our users to deal with those inconsistencies. A more sensible approach is to define a standard and flexible interface through a base class, allowing us to onboard various LLM providers and frameworks under a single, common API. That standard interface is the `BaseLLM` class, which we'll define in this section.

Text completion and chat are the two most standard LLM interaction modalities, which all LLM providers and frameworks support. Naturally, we'll also support these modes through the `BaseLLM` class via the methods `complete()` and `chat()`. There are a couple more interaction modes that `BaseLLM` supports, but I've omitted them for now to keep things focused. Figure 3.2 shows our LLM agent from before, but with the backbone LLM inheriting from the new `BaseLLM` class.

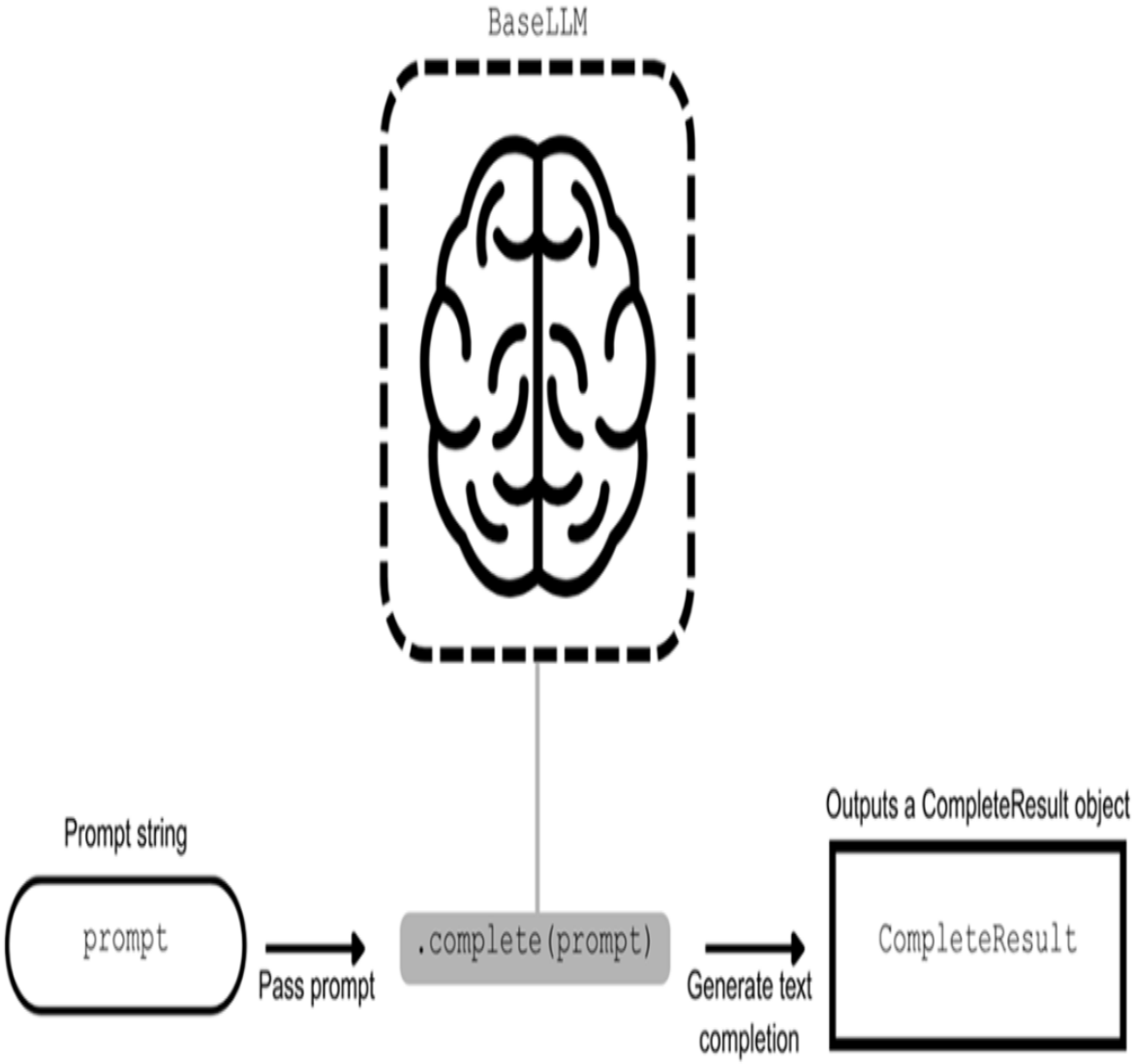
Figure 3.2 Our LLM agent with its backbone LLM and equipped tools. The backbone LLM is the Qwen3-7b model from the OllamaLLM class that we'll implement later in this chapter.


BaseLLM
<<async>> .complete()
Text completion mode of interaction
<<async>> .chat()
Chat mode of interaction
An async-first class, since all methods are asynchronous
inherits
LLM Agent
Backbone LLM
Tools
Qwen3-7b
function
RAG
MCP
code interpreter
MCP
plotter
websearch
MCP

The `complete()` method is designed for simple LLM text completion of a provided prompt, whereas `chat()` is for conversational dialogues with an LLM represented as a sequence of messages. Tool-calling with LLMs is typically handled through these chat interactions, which, as you'll soon see, is how we'll support it in our framework as well. Finally, it's important to note that `BaseLLM` is an async-first class, meaning all LLM interactions are executed asynchronously.

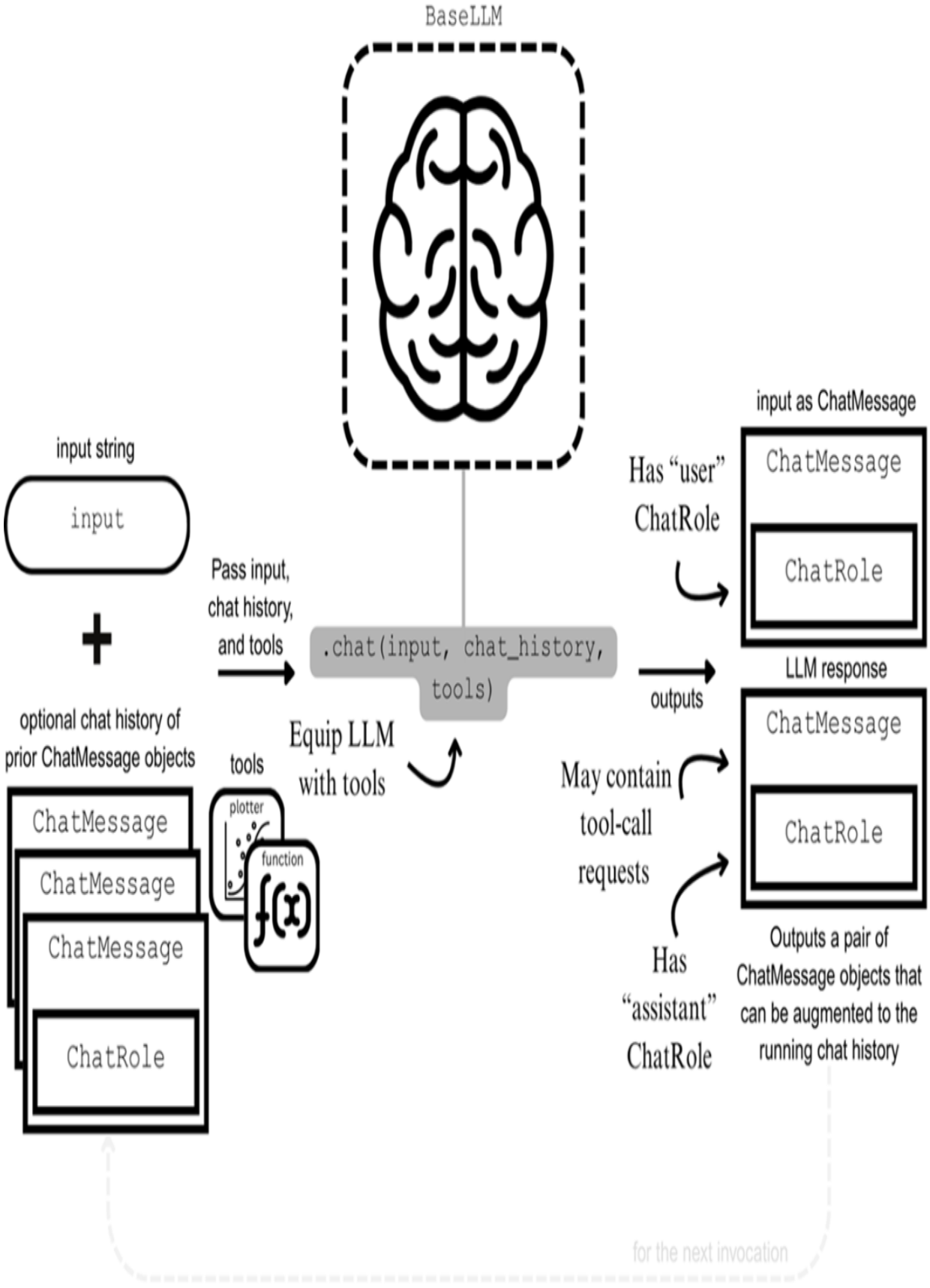
As mentioned earlier, we'll need a few new data structures to standardize the use of LLMs in our framework. First is `CompleteResult`, which we'll use to package the results of a `complete()` invocation. More specifically, we pass a prompt string as input to `complete()`, which then outputs a `CompleteResult` object containing the LLM's generated response. Figure 3.3 illustrates this process.

Figure 3.3 Supporting text completion with complete().



The next new data structure is `ChatMessage`, which facilitates chat interactions with LLMs. A `ChatMessage` object contains the content of the message and specifies its sender through another data structure, `ChatRole`. Figure 3.4 shows the process for chatting with LLMs through `chat()` and these data structures.

Figure 3.4 Supporting chat interactions with LLMs via chat().


BaseLLM
input as ChatMessage
input string
input
Has "user" ChatRole
ChatMessage
ChatRole
Pass input, chat history, and tools
.chat(input, chat_history, tools)
outputs
LLM response
optional chat history of prior ChatMessage objects
ChatMessage
ChatMessage
ChatMessage
ChatRole
tools
plotter
function
f(x)
Equip LLM with tools
May contain tool-call requests
ChatMessage
ChatRole
Has "assistant" ChatRole
Outputs a pair of ChatMessage objects that can be augmented to the running chat history
for the next invocation

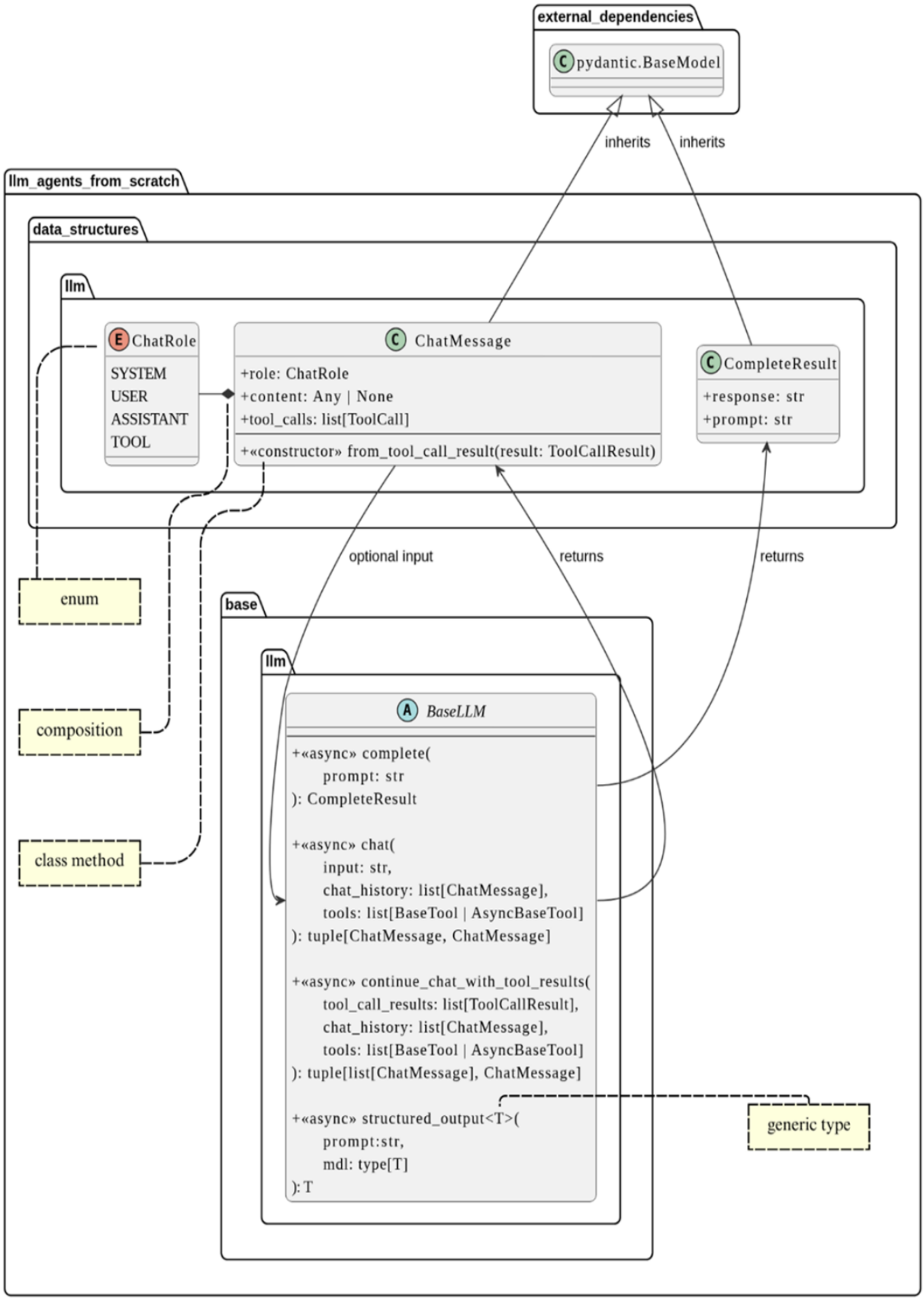
We invoke `chat()` with an input string, an optional chat history of `ChatMessage` objects, and a list of tools we want to equip the LLM with. The result of `chat()` is a new pair of `ChatMessage` objects. The first `ChatMessage` object is created from the user's input, whereas the second is created from the response generated by the LLM. If tool-call requests are made by the LLM, then they would be packaged in the second `ChatMessage` object. This returned pair can then be appended to the running chat history for the next `chat()` invocation.

note

I've elected to accept a simpler input type (i.e., `string`) for educational purposes and user convenience. Returning both the user input and LLM response as `ChatMessage` objects makes it easy to maintain a chat history under this design choice.

Now that we understand how the `BaseLLM` class and the new data structures can support the standard LLM interaction modes, let's go over their structural details. Figure 3.5 shows the UML class diagrams of `BaseLLM`, `CompleteResult`, `ChatMessage`, and `ChatRole`.

Figure 3.5 The UML class diagrams for BaseLLM and new data structures.


external_dependencies
pydantic.BaseModel
inherits
inherits
llm_agents_from_scratch
data_structures
llm
ChatRole
SYSTEM
USER
ASSISTANT
TOOL
ChatMessage
+role: ChatRole
+content: Any | None
+tool_calls: list[ToolCall]
+«constructor» from_tool_call_result(result: ToolCallResult)
CompleteResult
+response: str
+prompt: str
optional input
returns
returns
enum
base
llm
BaseLLM
+«async» complete(
prompt: str
): CompleteResult
+«async» chat(
input: str,
chat_history: list[ChatMessage],
tools: list[BaseTool | AsyncBaseTool]
): tuple[ChatMessage, ChatMessage]
+«async» continue_chat_with_tool_results(
tool_call_results: list[ToolCallResult],
chat_history: list[ChatMessage],
tools: list[BaseTool | AsyncBaseTool]
): tuple[list[ChatMessage], ChatMessage]
+«async» structured_output<T>(
prompt:str,
mdl: type[T]
): T
generic type
composition
class method

There are yet a few more new UML concepts introduced in figure 3.5. The first is the specification of `ChatRole` as an `enum` class, which is marked by the circle with the letter "E". Enumerations (or enums for short) are a programming type that specifies a finite set of named constants. Each instance can only be assigned one of the predefined values. The `enum` type is a perfect choice for the `ChatRole` class, which can only take on values of valid message senders: `SYSTEM`, `USER`, `ASSISTANT`, and `TOOL`. The next new UML concept is the composition relationship, indicated by a solid line with a filled diamond arrowhead. Composition relationships describe situations where one class is made up of other classes. The encompassing class is said to be composed of those other classes, which don't exist in a meaningful way on their own. Figure 3.5 shows that `ChatMessage` is composed of `ChatRole`, which aligns with our earlier discussion. It's worth noting that `ChatRole` doesn't meaningfully exist without the context of a `ChatMessage`.

Continuing our discussion of the `ChatMessage` class, you can see from figure 3.5 that it inherits from `pydantic.BaseModel` and has three attributes: `role`, `content`, and `tool_calls`. We'll discuss the meaning behind these attributes when we implement this data structure. In the meantime, you can also see that `ChatMessage` has one method, `from_tool_call_result()`, which has been marked with the `<<constructor>>` tag. This indicates that it's a constructor method, which in Python terminology is analogous to the `@classmethod` concept. You can interpret `from_tool_call_result`() as a method that creates a `ChatMessage` object from a `ToolCallResult` object, which, you'll see later, provides some convenience for us when implementing the final step to the tool-calling process.

The `CompleteResult` class also inherits from `pydantic.BaseModel` and has two attributes: `response` and `prompt`. The names of these attributes are self-documenting, but regardless, we'll discuss them when we implement `CompleteResult` in the next section.

Finally, the `BaseLLM` class is an abstract class with no attributes but four methods: `complete()`, `chat()`, `continue_chat_with_tool_results()`, and `structured_output()`. You can also see that `BaseLLM` is indeed async-first, with all four methods marked with the `<<async>>` tag to indicate their asynchronous nature. We've already discussed the standard LLM interaction modes, supported by `complete()` and `chat()`, and figure 3.5 reinforces how

`CompleteResult` and `ChatMessage` objects are used to facilitate these interactions.

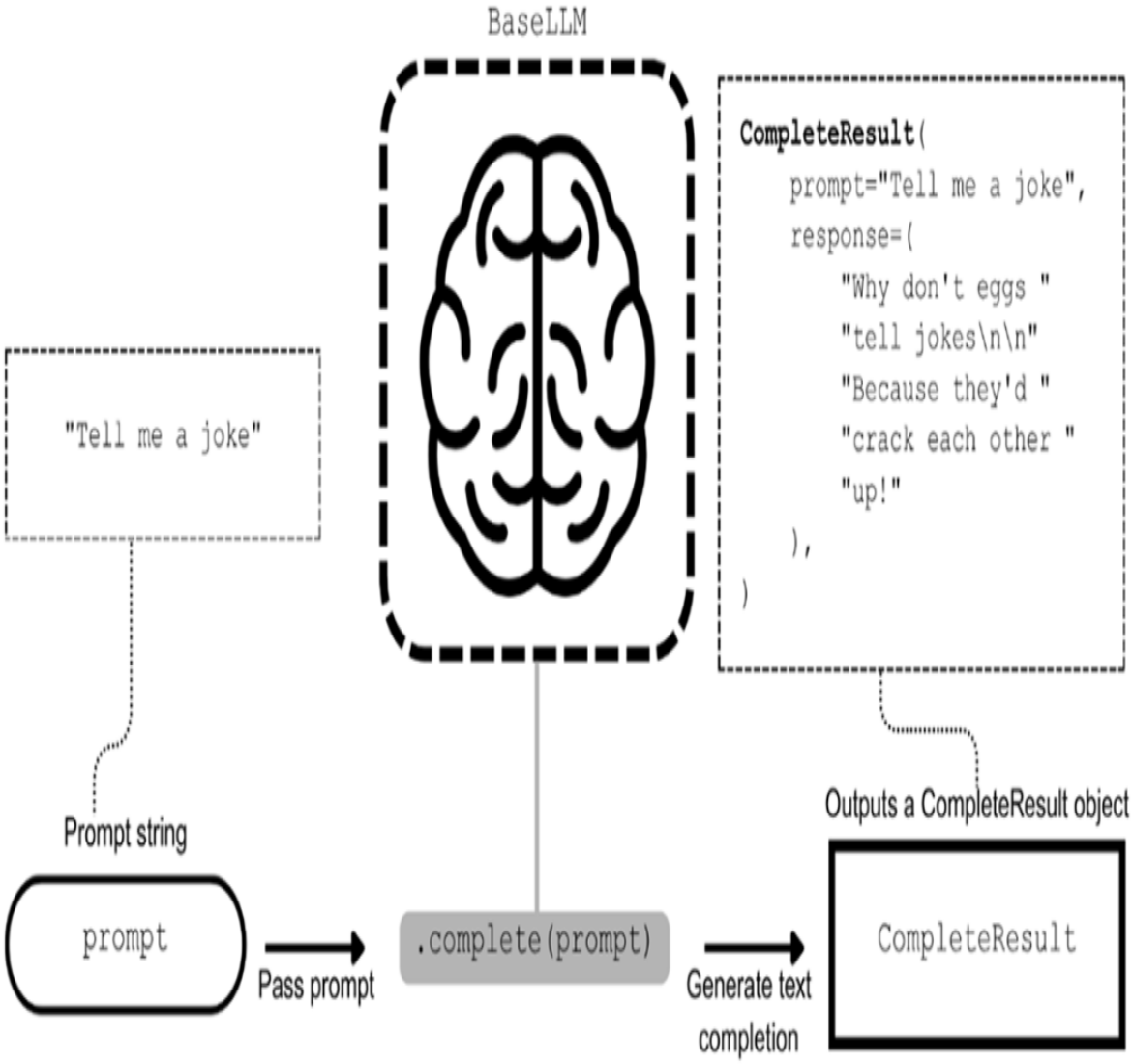
Let's now discuss the two new methods and the interaction modes they support. The first new method, `continue_chat_with_tool_results`(), extends `chat()` by providing a convenient way to submit tool results back to the LLM for synthesis and response. Since we're effectively continuing an existing chat interaction with this method, it also works with the already established `ChatMessage` and `ChatRole` data structures. On the other hand, `structured_output()` is designed for another useful LLM interaction mode, where we prompt LLMs to return their response in a pre-specified format, most often JSON. Figure 3.5 illustrates the use of a generic type, `T`, in the signature of `structured_output()`, which enables our users to specify their structured output using custom classes. We'll go over both methods in more detail when we implement the `BaseLLM` interface.

As we did in Chapter 2, we'll first implement the new data structures before implementing our main base class, `BaseLLM`.

3.1.1 Implementing CompleteResult, ChatMessage, and ChatRole

The first new data structure that we'll be adding to our framework is `CompleteResult`. It is a simple data structure that contains `prompt` and `response` attributes, both of which are of string type, as was shown in figure 3.5. The `prompt` attribute stores the input used to prompt the LLM, while the `response` attribute stores the LLM's generated response. Figure 3.6 shows the same text completion interaction process from earlier, but now includes an example input `prompt` and `CompleteResult` output object along with its attributes.

Figure 3.6 An example of a CompleteResult object that is returned from an invocation of complete().



The following code implements `CompleteResult`.

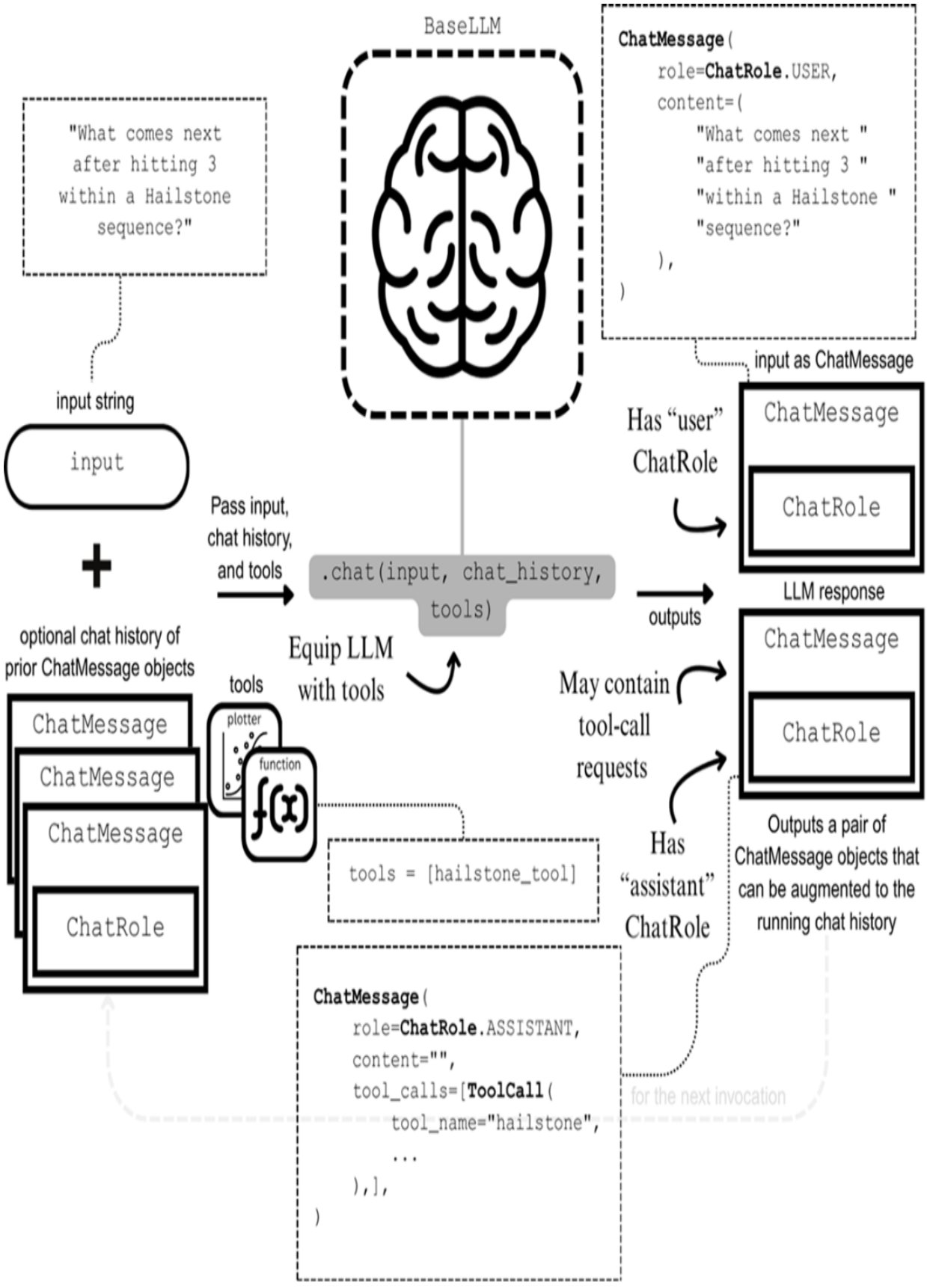
Listing 3.1 Implementing CompleteResult

```
# llm_agents_from_scratch/data_structures/llm.py
from pydantic import BaseModel
from typing_extensions import Self
from llm_agents_from_scratch.data_structures.tool import
ToolCall

class CompleteResult(BaseModel):
    """The LLM completion result data model.

    Attributes
        response: The completion response provided by the LLM.
```

```
        full_response: Input prompt and completion text.
    """

    response: str  #A
    prompt: str  #B
```

Let's next implement the data structures that facilitate the chat interactions: `ChatMessage` and `ChatRole`. Figure 3.7 shows the chat interaction first shown in figure 3.4, but this time layered with examples of these data structures.

Figure 3.7 Example ChatMessage and ChatRole objects within a chat() invocation.


BaseLLM
"What comes next after hitting 3 within a Hailstone sequence?"
ChatMessage(
role=ChatRole.USER,
content=(
"What comes next "
"after hitting 3 "
"within a Hailstone "
"sequence?"
),
)
input as ChatMessage
input string
input
Has "user" ChatRole
ChatMessage
ChatRole
Pass input, chat history, and tools
.chat(input, chat_history, tools)
outputs
LLM response
optional chat history of prior ChatMessage objects
ChatMessage
ChatMessage
ChatMessage
ChatRole
tools
plotter
function
f(x)
Equip LLM with tools
May contain tool-call requests
ChatMessage
ChatRole
tools = [hailstone_tool]
Has "assistant" ChatRole
Outputs a pair of ChatMessage objects that can be augmented to the running chat history
ChatMessage(
role=ChatRole.ASSISTANT,
content="",
tool_calls=[ToolCall(
tool_name="hailstone",
...
),],
)
for the next invocation

As you can see, the user's input is returned as a `ChatMessage` with the `USER` role, whereas the LLM's response is a `ChatMessage` with the `ASSISTANT` role. In this example, the LLM's response carries a `ToolCall` object, and in these cases, the `content` of the `ChatMessage` object is an empty string.

Standardized message roles

The role definitions for messages have been standardized to some extent across various LLM providers and frameworks and can be described as follows. The `SYSTEM` role is reserved for setting the general context for the LLM or LLM agent, such as defining its role in the upcoming chat interaction. The `USER` role is reserved for messages sent by the user, whereas the `ASSISTANT` role is meant for messages from the LLM. Finally, the `TOOL` role is for messages that carry the tool results sent back to the LLM.

For convenience, `ChatMessage` objects can also be derived from a `ToolCallResult` via the constructor method, `from_tool_call_result()`. This method returns a `ChatMessage` object with the `TOOL` role and whose `content` is a string serialization of the `ToolCallResult` object.

The implementations for `ChatMessage` and `ChatRole` are shown in the following code.

Listing 3.2 Implementing ChatMessage and ChatRole

```
# llm_agents_from_scratch/data_structures/llm.py
from pydantic import BaseModel
from typing_extensions import Self
from llm_agents_from_scratch.data_structures.tool import
ToolCall

…  #A

class ChatRole(str, Enum):
    """Roles for chat messages."""

    USER = "user"
    ASSISTANT = "assistant"
    SYSTEM = "system"
    TOOL = "tool"

class ChatMessage(BaseModel):
```

```
    """The chat message data model.

    Attributes:
        role: The role of the message.
        content: The content of the message.
        tool_calls: Tool calls associated with the message.
    """

    model_config = ConfigDict(arbitrary_types_allowed=True)
    role: ChatRole
    content: str
    tool_calls: list[ToolCall] | None = None  #B

    @classmethod  #C
    def from_tool_call_result(
        cls,
        tool_call_result: ToolCallResult
    ) -> Self:
        """Create a ChatMessage from a ToolCallResult."""
        return cls(
            role=ChatRole.TOOL,  #D
            content=tool_call_result.model_dump_json(indent=4),
#E
        )
```

3.1.2 Implementing BaseLLM

We're now ready to start implementing the `BaseLLM` class. As you saw from the UML class diagrams in figure 3.5, `BaseLLM` is an abstract class that has no attributes but four methods: `complete()`, `chat()`, `continue_chat_with_tool_results()`, and `structured_output()`.

We'll start with `complete()` and `chat()`, both of which will be marked as abstract and whose input and output types we've already covered in great detail. The following listing shows their implementation.

Listing 3.3 Implementing BaseLLM: chat() and complete()

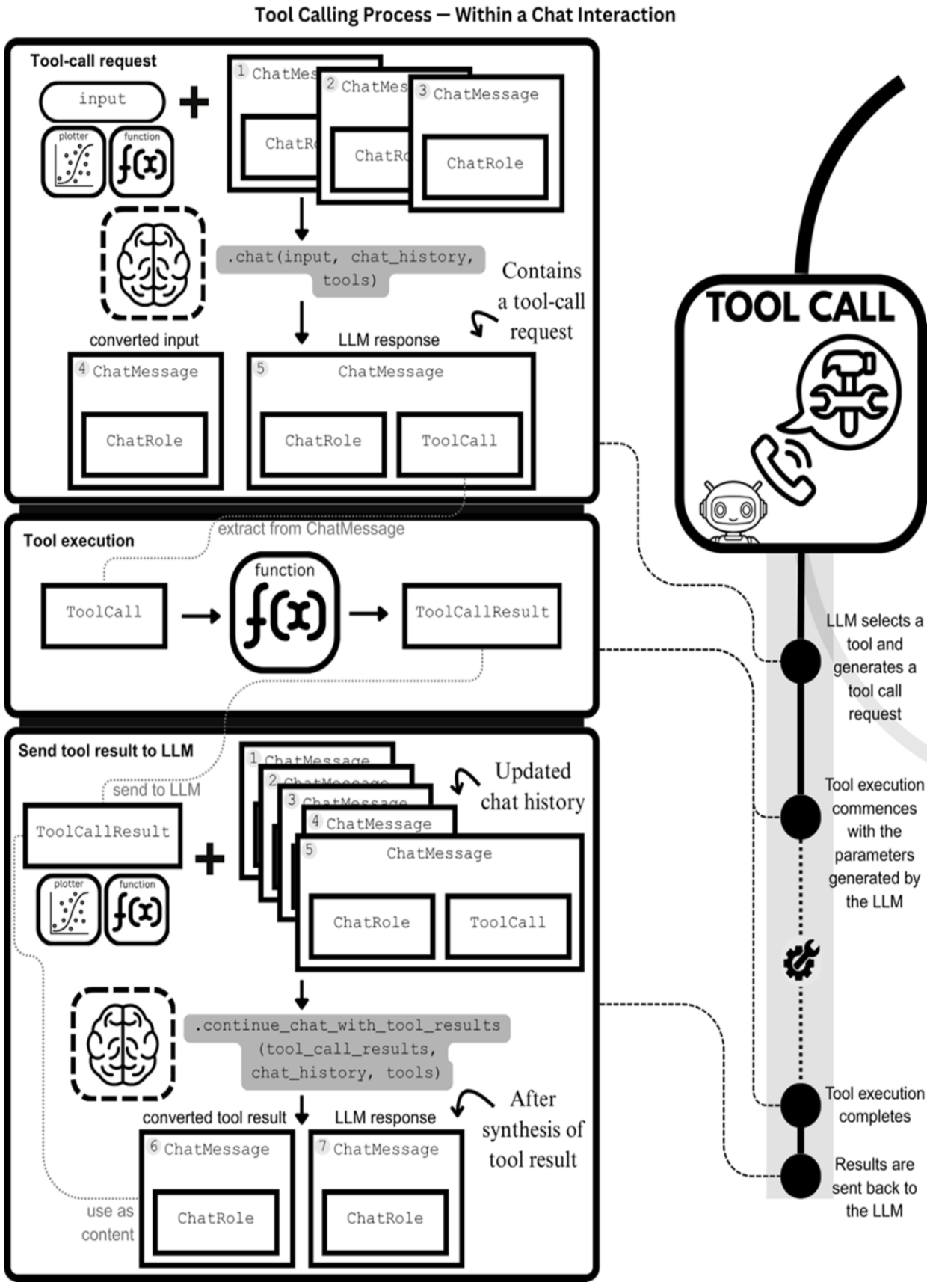
```
# llm_agents_from_scratch/base/llm.py
from abc import ABC, abstractmethod
from llm_agents_from_scratch.base.tool import AsyncBaseTool,
BaseTool
from llm_agents_from_scratch.data_structures import (
    ChatMessage,
```

```
    CompleteResult,
    ToolCallResult,
)
from typing import Any, Sequence

Tool: TypeAlias = BaseTool | AsyncBaseTool  #A

class BaseLLM(ABC):
    """Base LLM Class."""

    @abstractmethod
    async def complete(
        self,
        prompt: str,  #B
        **kwargs: Any
    ) -> CompleteResult:  #C
        """Text Complete."""

    @abstractmethod
    async def chat(
        self,
        input: str,  #D
        chat_messages: Sequence[ChatMessage] | None = None,  #E
        tools: Sequence[Tool] | None = None,  #F
        **kwargs: Any,
    ) -> tuple[ChatMessage, ChatMessage]:  #G
        """Chat interface."""
```

As with `BaseTool` in Chapter 2, subclasses of `BaseLLM` will need to provide implementations for all methods marked as abstract.

Let's now move on to specifying our convenient extension of `chat()` used for returning tool call results back to the LLM: `continue_chat_with_tool_results()`. Before writing any code, let's revisit the tool-call process one more time. Figure 3.8 illustrates the familiar tool-calling process—first introduced in Chapter 1—but now projected into an LLM chat interaction.

Figure 3.8 The familiar tool-calling process, now shown within a chat() invocation.


Tool Calling Process — Within a Chat Interaction
Tool-call request
input
plotter
function
1 ChatMes
2 ChatMes
3 ChatMessage
ChatRole
.chat(input, chat_history, tools)
Contains a tool-call request
converted input
LLM response
4 ChatMessage
ChatRole
5 ChatMessage
ChatRole
ToolCall
Tool execution
extract from ChatMessage
ToolCall
function
ToolCallResult
Send tool result to LLM
send to LLM
ToolCallResult
Updated chat history
4 ChatMessage
5 ChatMessage
ChatRole
ToolCall
.continue_chat_with_tool_results (tool_call_results, chat_history, tools)
converted tool result
LLM response
After synthesis of tool result
6 ChatMessage
7 ChatMessage
ChatRole
use as content
TOOL CALL
LLM selects a tool and generates a tool call request
Tool execution commences with the parameters generated by the LLM
Tool execution completes
Results are sent back to the LLM

A tool-call request, within a chat interaction, is represented by a `ChatMessage` that contains `ToolCall` objects in its `tool_calls` attribute. Note that only `ChatMessage` objects with the `ASSISTANT` role can carry tool-call requests. `ToolCall` objects need to be extracted from such messages and passed to the `__call__()` method of the selected tool for execution. After the tool's execution, we need to send the produced `ToolCallResult` object(s) back to the LLM for synthesis and response. To do so, we utilize our convenient chat extension method, `continue_chat_with_tool_results()`, which should first create `ChatMessage` objects from the supplied `ToolCallResult` objects before passing them on to the LLM.

The inputs for `continue_chat_with_tool_results()` are almost identical to those for `chat()`, with the only difference being that, rather than an input string being passed to `chat()`, `ToolCallResult` objects are passed instead.

The output of `continue_chat_with_tool_results()`, like for `chat()`, is a tuple of `ChatMessage` objects. The first element is the list of `ChatMessage` objects derived from the input `ToolCallResult` objects, each with the `TOOL` role. The second element is the LLM's response to the tool-call results as a `ChatMessage`. This maintains the same pattern for updating your chat history—append both elements of the returned tuple to your running chat history, whether you're using `chat()` or `continue_chat_with_tool_results()`.

NOTE

We could forego the use of `continue_chat_with_tool_results()` and use `chat()` directly instead. This process would involve having to convert the `ToolCallResult` objects manually into `ChatMessage` objects with the appropriate `TOOL` role and passing them along with the updated chat history to the next `chat()` invocation. However, for both convenience and educational purposes, I've elected to include this method to make the entire tool-call process within an LLM chat interaction more explicit.

The following code shows the implementation of `continue_chat_with_tool_results()`.

Listing 3.4 Implementing BaseLLM: continue_chat_with_tool_results()

```
# llm_agents_from_scratch/base/llm.py
from abc import ABC, abstractmethod
from llm_agents_from_scratch.base.tool import AsyncBaseTool,
BaseTool
from llm_agents_from_scratch.data_structures import (
    ChatMessage,
    CompleteResult,
    ToolCallResult,
)
from typing import Any

class BaseLLM(ABC):
    """Base LLM Class."""
    …  #A

    @abstractmethod
    async def continue_chat_with_tool_results(
        self,
        tool_call_results: Sequence[ToolCallResult],  #B
        chat_history: Sequence[ChatMessage],  #C
        tools: Sequence[Tool] | None = None,
        **kwargs: Any,
    ) -> tuple[list[ChatMessage], ChatMessage]:  #D
        """Continue a chat submitting tool call results."""
```

We'll now move on to implementing the final method for `BaseLLM`: `structured_output()`. Before writing any code, let's first discuss the motivation for including this method in our interface in the first place.

As text generators, LLMs can also produce structured outputs. That is, we can elicit LLMs to generate text that conforms to a pre-specified structured format, most typically JSON. Structured outputs simplify downstream processing by reducing the need to implement logic that is often brittle and unreliable for extracting the required elements from raw output strings. Let's consider the simple example below, which demonstrates the brittleness of unstructured, raw outputs.

Tell me a joke from any of these three subjects: math, physics, and biology. Also, include the subject of the joke.

An unstructured response

Here's one:

Why did the DNA go to therapy? Because it was feeling a little twisted! (Biology)

From this lone response, we can easily extract the joke's subject, biology. But even just prompting the LLM a second time with the same prompt as before can lead to a drastically different form of output.

Another LLM unstructured response

Here's one:

Why did the math book look so sad? Because it had too many problems.

Subject: Math

Like with the first output, it'd be easy to extract that the subject is math upon manual inspection. However, if we wanted to build robust code that depends on the accurate parsing of this free-form output, we're likely setting ourselves up for failure. A better approach is to prompt the LLM to produce its output in a structured format, as illustrated next.

Tell me a joke from any of these three subjects: math, physics, and biology. Also, include the subject of the joke.

Return your output in the format provided below:

```
‘{
    “subject”: …,
    “joke”: …,
}’
```

Structured response

```
{
    "subject": "math",
    "joke": "Why did the math book look so sad? Because it had
too many problems."
}
```

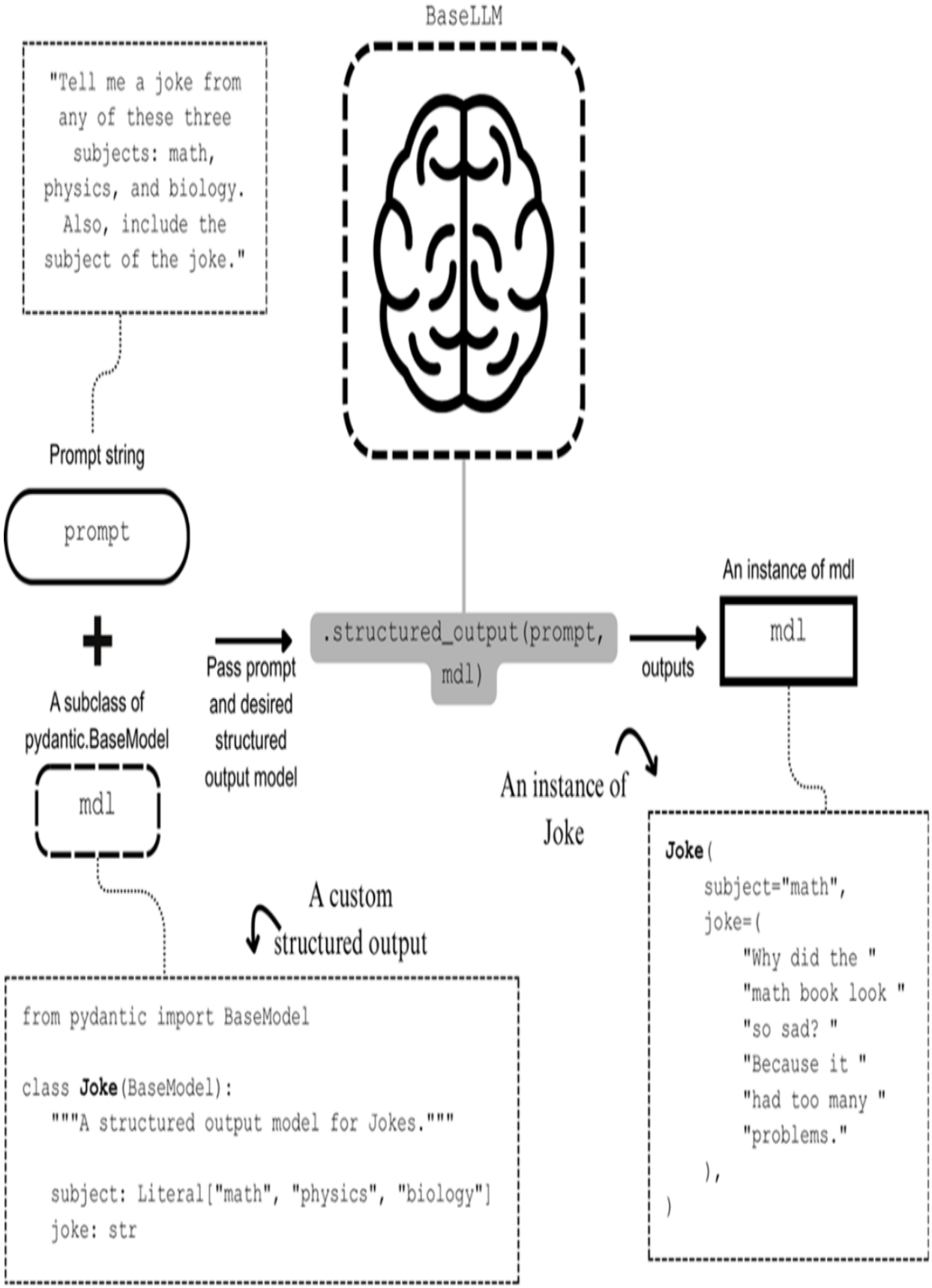
This structured output is much easier to work with than the free-form versions and would allow us to build downstream logic that depends on

these formatted outputs. Figure 3.9 illustrates how this example can be executed through the `structured_output()` method.

note

Similar to tool calling, LLM inference providers and frameworks often provide a dedicated API for structured outputs. These are usually designed such that the user only needs to supply the desired output schema, typically JSON, and can offload the job of instructing the LLM to produce output in this format to the service provider (or framework). In contrast, in our example, we manually elicited the LLM to produce output in our desired format.

Figure 3.9 The structured output LLM interaction. An instruction prompt and the desired output format, mdl, are passed as the inputs to structured_output(). The LLM generates a response that can then be used to create an instance of mdl, a subclass of pydantic.BaseModel.


BaseLLM
"Tell me a joke from any of these three subjects: math, physics, and biology. Also, include the subject of the joke."
Prompt string
prompt
+
A subclass of pydantic.BaseModel
mdl
Pass prompt and desired structured output model
.structured_output(prompt, mdl)
outputs
An instance of mdl
mdl
An instance of Joke
A custom structured output
from pydantic import BaseModel
class Joke(BaseModel):
"""A structured output model for Jokes."""
subject: Literal["math", "physics", "biology"]
joke: str
Joke(
subject="math",
joke=(
"Why did the "
"math book look "
"so sad? "
"Because it "
"had too many "
"problems."
),
)

Now that we understand the rationale for this `structured_output()`, let's focus on its implementation. As you saw in Figure 3.5, the signature of `structured_output()` specifies a `prompt` string and a `mdl` parameter as inputs, where `mdl` allows us to specify the desired output format. This `mdl` parameter, as you might recall from our earlier discussion, depends on a generic `T`. We'll bind `T` to the `pydantic.BaseModel` class. Doing so means that we must define our desired structured output type via a subclass of `pydantic.BaseModel`.

The `structured_output()` method returns an instance of `T`. In other words, it returns an instance of our desired structured output class. For increased clarity, we refer to `T` as `StructuredOutputType` in our implementation, as shown in the following code.

Listing 3.5 Implementing BaseLLM: structured_output()

```
# llm_agents_from_scratch/base/llm.py
from abc import ABC, abstractmethod
from typing import Any, Sequence, TypeVar
from pydantic import BaseModel
from llm_agents_from_scratch.base.tool import AsyncBaseTool,
BaseTool
   from llm_agents_from_scratch.data_structures import (
    ChatMessage,
    CompleteResult,
    ToolCallResult,
)

StructuredOutputType = TypeVar("StructuredOutputType",
bound=BaseModel)  #A

class BaseLLM(ABC):
    """Base LLM Class."""
    …  #B

    @abstractmethod
    async def structured_output(
        self,
        prompt: str,
        mdl: type[StructuredOutputType],  #C
        **kwargs: Any,
    ) -> StructuredOutputType:  #D
        """Structured output interface for returning
~pydantic.BaseModels.
```

NOTE

Binding the generic `StructuredOutputType` to `pydantic.BaseModel` is an implementation detail that lets us fully leverage the powerful validation checks offered by the `pydantic` library.

Let's take our joke example from before and build a structured output model for it. This structured output model is what we'd supply for the `mdl` parameter in a `structured_output()` invocation. As you now know, this model needs to inherit from `pydantic.BaseModel` as shown in the following code.

```
# Included in examples/ch03.ipynb  #A
from typing import Literal
from pydantic import BaseModel

class Joke(BaseModel):  #B
    """A structured output model for Jokes."""

    subject: Literal["math", "physics", "biology"]
       joke: str
```

To demonstrate an actual invocation of `structured_output()` using this `Joke` class, we'll first need a proper subclass of `BaseLLM` that has an implementation for this method. This is precisely where we're heading in the next section.

Tool calling as structured output and vice versa

Tool calling can also be viewed as a structured output. We could even use the `structured_output()` method to get the LLM to produce a tool-call request JSON specifying the selected tool's name and the argument values to invoke the tool with.

However, tool-calling APIs offered by LLM service providers and frameworks have ready-made prompt templates that instruct the LLM to use the specified tools only if necessary. They also have prompt templates for instructing an LLM to synthesize and respond to tool-call results whenever it receives them. For these reasons, it's better to rely on these native tool-calling APIs versus trying to make it work with structured outputs.

It's also worth mentioning that it's entirely possible to implement structured output interactions as a tool call. That is, by defining a tool that generates the desired structured format and forcefully instructing the LLM to use it.

3.2 OllamaLLM: a subclass of BaseLLM

Now that we've specified the way we'll work with LLMs in our framework, through the `BaseLLM` class, let's implement one. In this section, we'll build an integration with the Ollama LLM inference framework. This integration will enable us to use any open-source LLM supported by Ollama. We'll implement the `OllamaLLM` class, which interacts with an Ollama service, typically running locally on your machine.

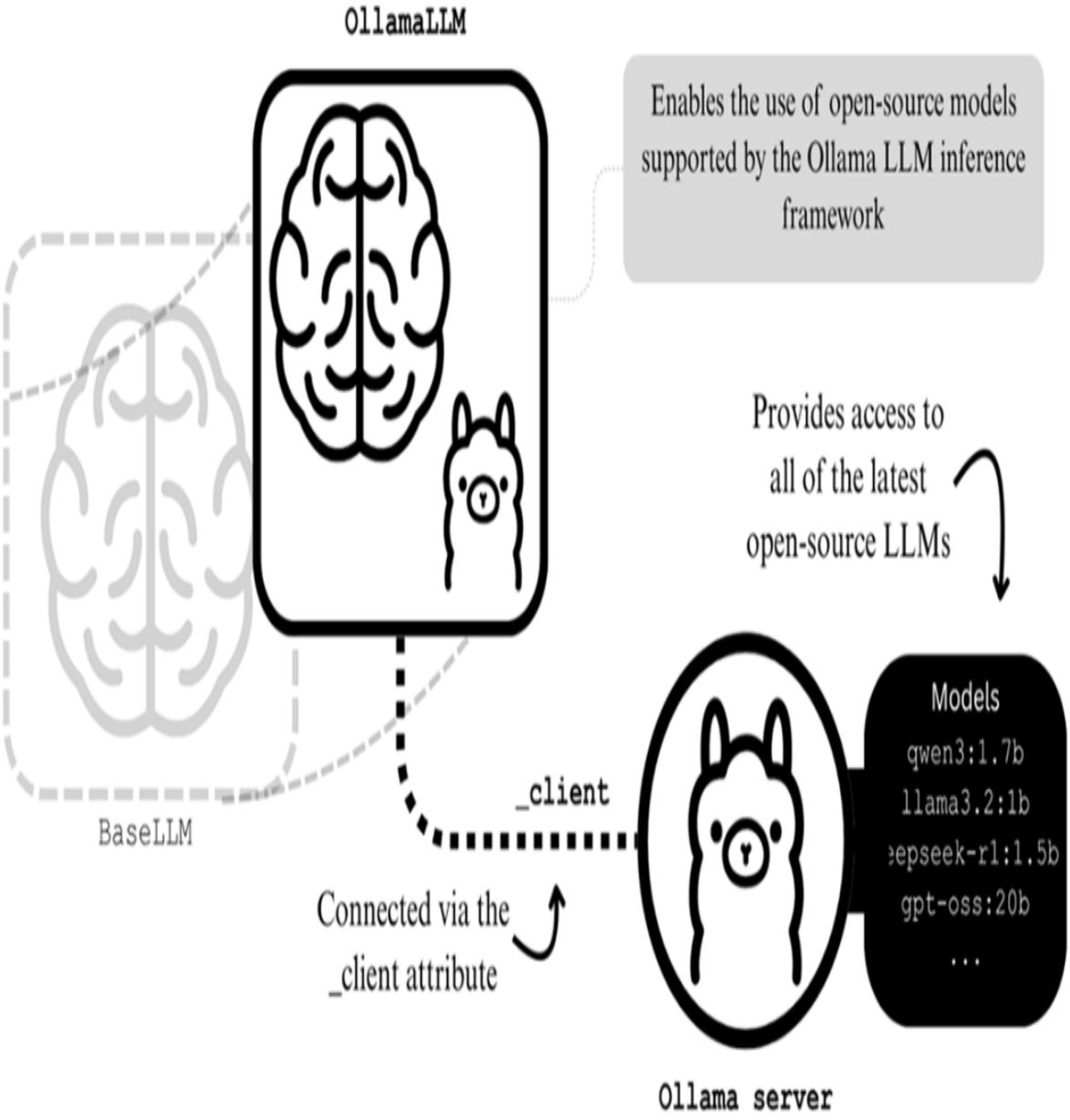
Implementing `OllamaLLM` requires using the `ollama` Python library. Once implemented, we'll run through examples for interacting with an LLM via `chat()`, `complete()`, `continue_chat_with_tool_results()`, and `structured_output()`.

note

Building an integration like this often requires reading documentation, source code, and other resources for the library. In writing this section, I referenced the official documentation and even the source code of the `ollama` library to determine how to build this integration effectively. While you won't need to do this for the current Ollama integration, as we'll walk through the completed code together, it may be helpful to know what resources were referenced.

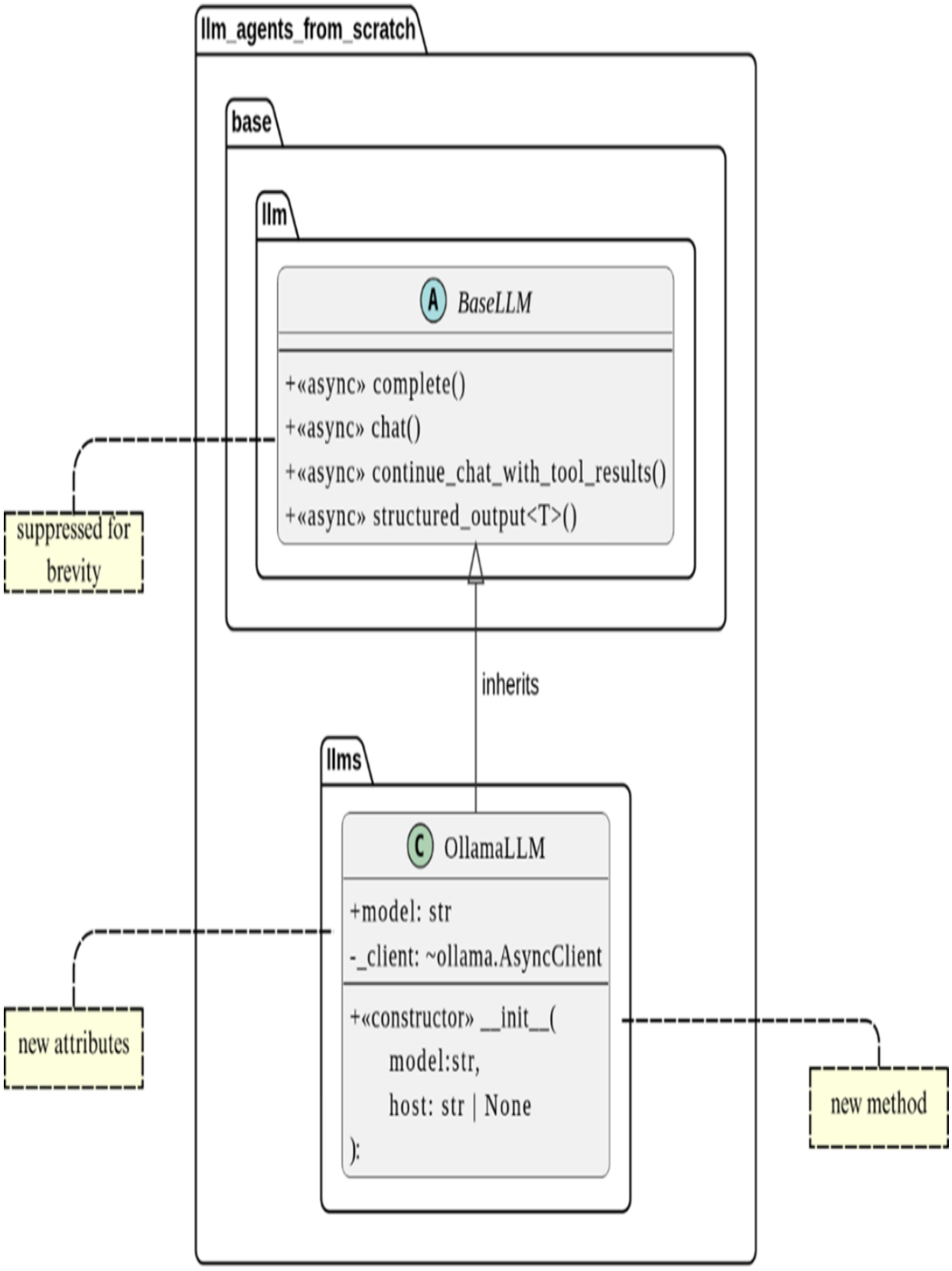
Figure 3.10 shows our Ollama integration through `OllamaLLM`, a subclass of `BaseLLM`.

Figure 3.10 Integrating with the Ollama LLM inference framework. The OllamaLLM connects to a running Ollama server, often running locally on your machine, to work with any of the supported open-source LLMs.



In addition to the methods and attributes of `BaseLLM`, the `OllamaLLM` class introduces a `_client` attribute for interacting with a running Ollama service. The complete structure of `OllamaLLM` is outlined in its UML class diagram shown in figure 3.11.

Figure 3.11 UML class diagram for OllamaLLM.


llm_agents_from_scratch
base
llm
BaseLLM
+«async» complete()
+«async» chat()
+«async» continue_chat_with_tool_results()
+«async» structured_output<T>()
suppressed for brevity
inherits
llms
OllamaLLM
+model: str
-_client: ~ollama.AsyncClient
+«constructor» __init__(
model:str,
host: str | None
):
new attributes
new method

You can see that the `OllamaLLM` class adds two new attributes and one method. The `model` attribute specifies which supported LLM to use, while `_client` connects to a running Ollama service and is of the type `AsyncClient` from the `ollama` library. Finally, `__init__()` is used for initializing objects of `OllamaLLM`.

NOTE

To run the code examples in this section, you'll need Ollama installed on your local machine with its service running. To download Ollama, follow the instructions at https://ollama.com/download. After installation, a service may launch automatically. If not, you can start one by opening a terminal and running the command `ollama serve`.

3.2.1 Implementing OllamaLLM

We're going to implement this integration step-by-step, starting with the implementation of the `__init__()` method.

As shown in figure 3.11, the `__init__()` method takes in `model` and `host` parameters as inputs. The `model` parameter is a string type that specifies the LLM we'd like to use, such as `llama3.2:1b`. The optional `host` parameter specifies the address for the Ollama service that we'd like to interact with. If no `host` is provided, the default Ollama service address will be used.

note

Running `ollama serve` on your machine launches a service at the default address http://127.0.0.1:11434. If you don't provide a value for `host` when initializing an `OllamaLLM` instance, this default address will be used.

Within the `__init__()` method, we'll use the provided parameters to set the `model` and `_client` attributes of the `OllamaLLM` instance. The following code shows the implementation of `__init__()`.

Listing 3.6 Implementing OllamaLLM: __init__()

```
   # llm_agents_from_scratch/llms/ollama/llm.py
from typing import Any
from ollama import AsyncClient  #A
from llm_agents_from_scratch.base.llm import BaseLLM


class OllamaLLM(BaseLLM):
    """Ollama LLM class."""

    def __init__(
        self,
        model: str,  #B
        host: str | None = None,
        *args: Any,
        **kwargs: Any,
    ) -> None:
        """Create an OllamaLLM instance.

        Args:
            model (str): The name of the LLM model.
            host (str | None): Host of running Ollama service.
Defaults to
                None.
            *args (Any): Additional positional arguments.
            **kwargs (Any): Additional keyword arguments.
        """
        super().__init__(*args, **kwargs)
        self.model = model
        self._client = AsyncClient(host=host)  #C
```

To see this `__init__()` method in action, let's create an instance of our `OllamaLLM` class that connects to the default host and uses the 3-billion parameter version of the Qwen 2.5 LLM. The following code snippet demonstrates how to do that.

```
# Included in examples/ch03.ipynb  #A
from llm_agents_from_scratch.llms.ollama import OllamaLLM

llm = OllamaLLM(model="qwen2.5:3b")
```

note

The remainder of this chapter uses the `qwen2.5:3b` model, which Ollama supports. You will need to pull this model from Ollama first to run the

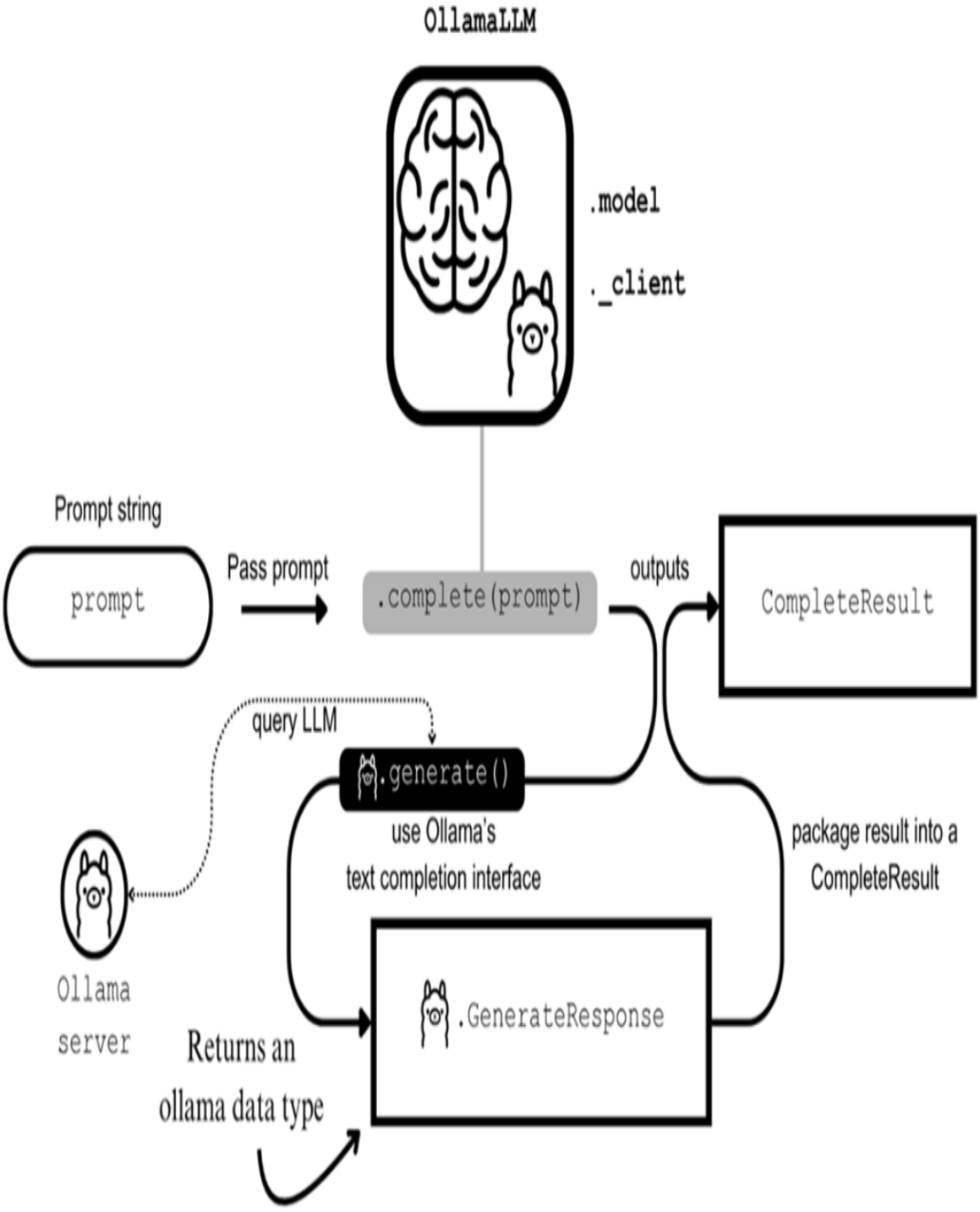
remaining code snippets by using the terminal command `ollama pull qwen2.5:3b`.

With our `__init__()` method established, let's now focus on implementing all the abstract methods required by `BaseLLM`, starting with the `complete()` method.

The logic for our `complete()` method will have us using the instance's `_client` attribute to interact with our running Ollama service. More specifically, the `ollama.AsyncClient` class contains a method named `generate()` that is Ollama's interface for supporting the LLM completion interaction mode.

The `generate()` method of `ollama.AsyncClient` requires two parameters: `model` and `prompt`. We provide the name of the model to use by passing along our instance's `model` attribute. For the `prompt` parameter, we simply forward the `prompt` variable from the outer `complete()` call. The result of `generate()` is an Ollama data type, which we'll use to derive a `CompleteResult` object that we'll return as the final output of `complete()`. Figure 3.12 shows how `complete()` integrates with the `ollama` library through the `generate()` method.

Figure 3.12 Integrating with Ollama's text completion interface. A complete() invocation of OllamaLLM invokes the generate() method of the Ollama library and returns an Ollama data type from which we derive the final return type, CompleteResult.



The following code provides the implementation of `complete()` for `OllamaLLM`.

Listing 3.7 Implementing OllamaLLM: complete()

```
# llm_agents_from_scratch/llms/ollama/llm.py
from typing import Any
from ollama import AsyncClient
from llm_agents_from_scratch.base.llm import BaseLLM
from llm_agents_from_scratch.data_structures import (
    CompleteResult,
)


class OllamaLLM(BaseLLM):
    """Ollama LLM class."""
    …  #A

    async def complete(self, prompt: str, **kwargs: Any) ->
CompleteResult:
        """Complete a prompt with an Ollama LLM.

        Args:
            prompt (str): The prompt to complete.
            **kwargs (Any): Additional keyword arguments.

        Returns:
            CompleteResult: The text completion result.
        """
        response = await self._client.generate(  #B
            model=self.model,  #C
            prompt=prompt,  #D
            **kwargs,
        )
        return CompleteResult(
            response=response.response,  #E
            prompt=prompt,
        )
```

Let's go over a demonstration of our now-implemented `complete()` method. We'll use the `qwen2.5:3b` LLM and have it tell us a joke.

```
# Included in examples/ch03.ipynb  #A
import asyncio
from llm_agents_from_scratch.llms.ollama import OllamaLLM

async def main():
    llm = OllamaLLM(model="qwen2.5:3b")
    response = await llm.complete("Tell me a joke.")
    print(response)

asyncio.run(main())
```

You learned that `complete()` returns a `CompleteResult` object that contains a `response` and `prompt` attribute. Therefore, `print(response)` outputs the string representation of this class, which should resemble the following code.

```
response="Sure! Here's one for you:\n\nWhy don't scientists
trust atoms?\n\nBecause they make up everything!" prompt='Tell
me a joke.'
```

NOTE

Async methods need to be run within an async event loop. The `asyncio.run()` method creates a new event loop and runs the provided `Coroutine`. When using Jupyter notebooks, there is already a running event loop, which means async methods can be awaited directly without using `asyncio.run()`.

With the `complete()` method implemented, rather than implementing `chat()` next, let's implement `structured_output()`. The reason for this is that we'll use the same "tell me a joke" example that we just covered, but using our `Joke` model from earlier as our desired structured output format.

To implement `structured_output()`, we'll need to use Ollama's structured output interface, much like how our implementation of `complete()` relied on Ollama's completion interface, `generate()`. As it turns out, Ollama's structured output interface is facilitated by their chat interface. To not confuse Ollama's chat interface with that of our framework, I'll refer to Ollama's as `ollama.chat()`.

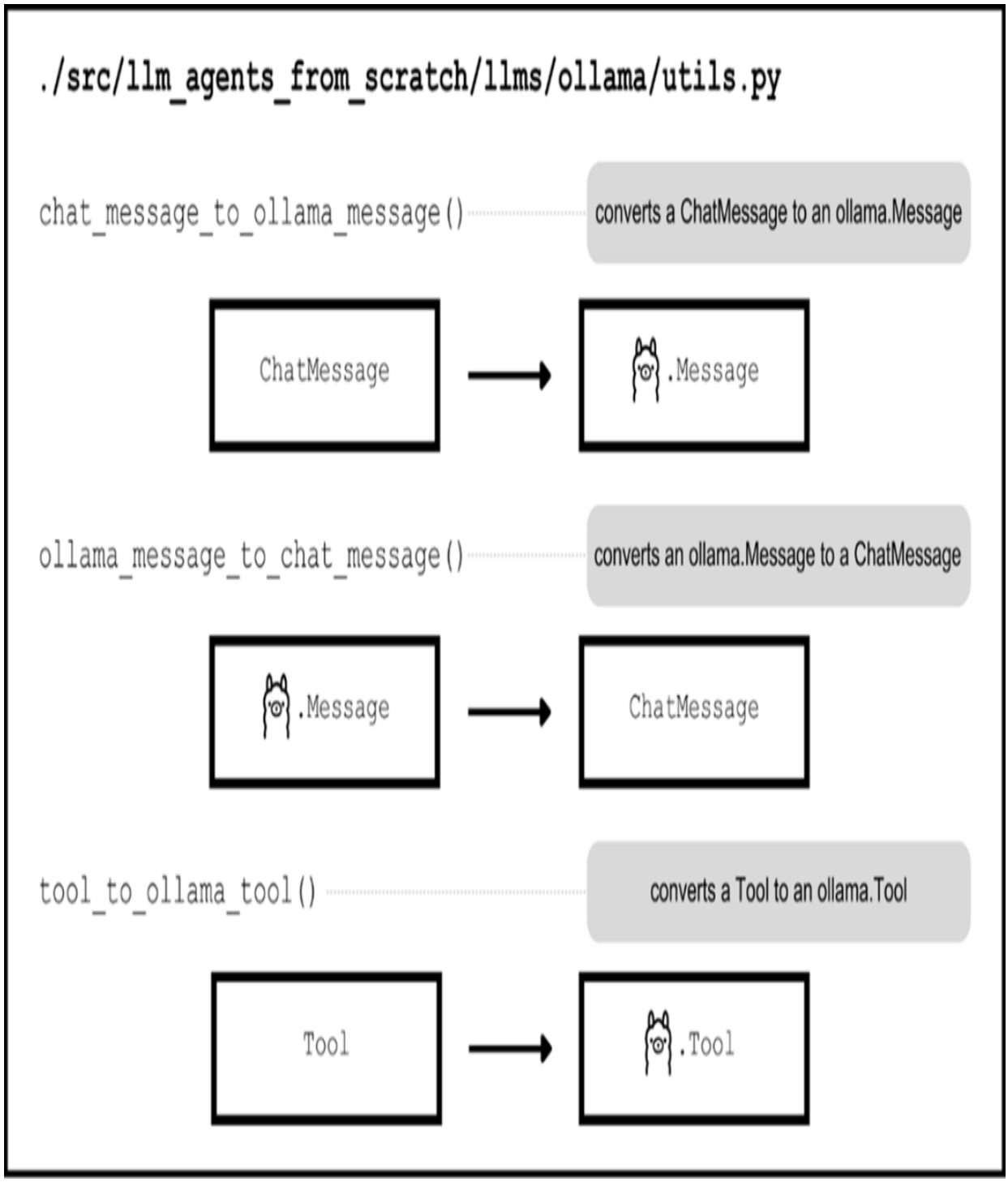
The `ollama.chat()` method allows for the optional specification of a `format` parameter, where users can supply their desired structured output as a JSON Schema. You learned about JSON Schemas in the previous chapter, and you may recall they can be produced for `pydantic.BaseModel` subclasses through the `model_json_schema()` method. We'll rely on this as well in our implementation of `structured_output()` since the parameter `mdl` is a type that is bound to `pydantic.BaseModel`.

Before we show the code that implements `structured_output()`, we need to discuss an important activity that typically takes place when building integrations with LLM frameworks like Ollama. That is, developing these

integrations often requires converting between the data types of our framework and those of the library we want to support, and vice versa. We saw a bit of this in our previous implementation of `complete()`, when we had to extract the `response` attribute from the returned Ollama data type to create our `CompleteResult`. For our Ollama integration, we'll use three utility functions that explicitly convert the data structures of our framework to the corresponding Ollama data type equivalents, and vice versa for the message data type. The three utility methods are listed below and shown in figure 3.13:

- `chat_message_to_ollama_message()`
- `ollama_message_to_chat_message()`
- `tool_to_ollama_tool()`

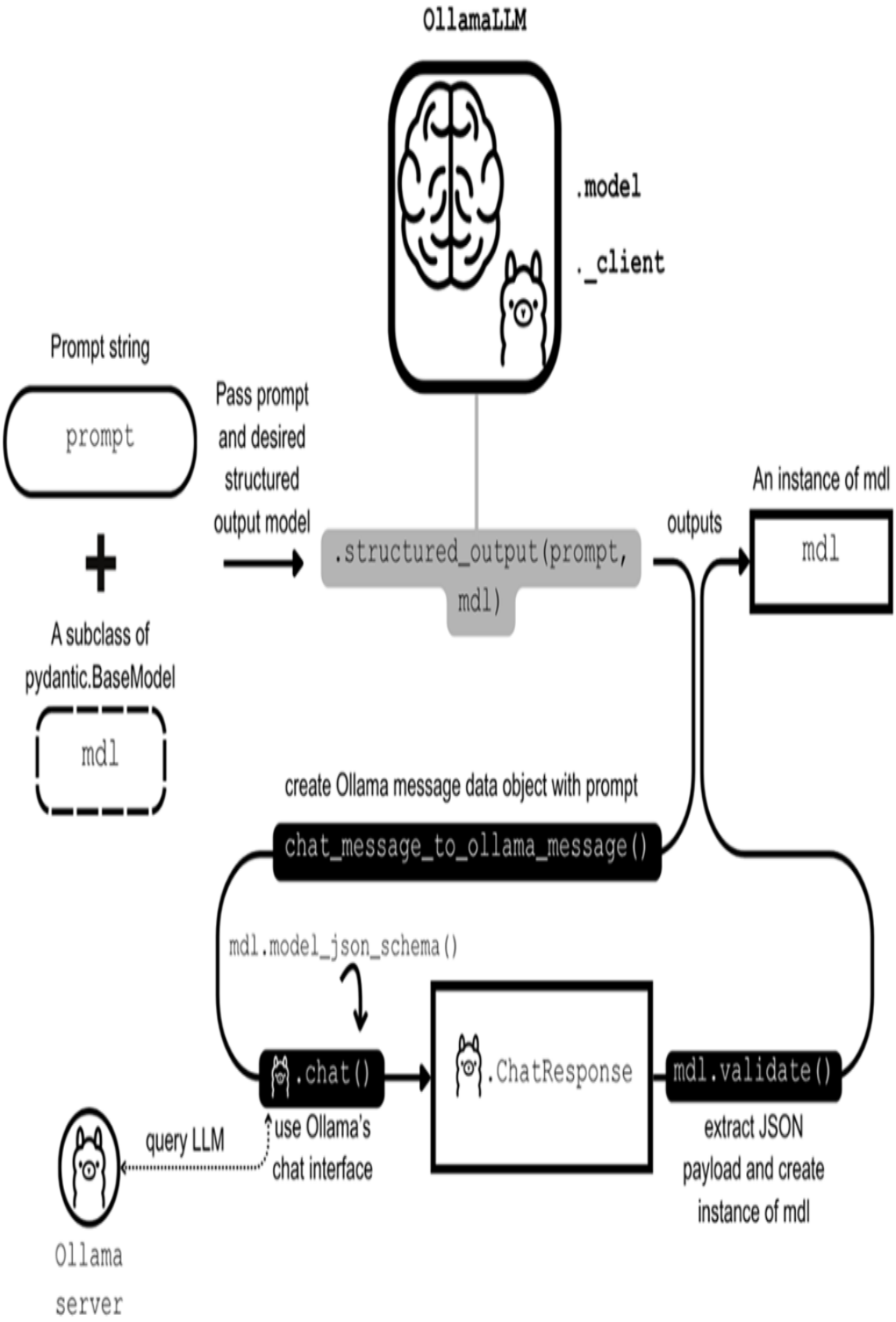
Figure 3.13 Three utility functions for converting between llm-agents-from-scratch data types and corresponding Ollama equivalent data types.



For brevity, and since the implementations of these utility functions are not core to understanding how we'll build `OllamaLLM`, I've not included them here. Interested readers can view their complete implementations at: https://github.com/nerdai/llm-agents-from-scratch/blob/main/src/llm_agents_from_scratch/llms/ollama/utils.py.

We now have all the pieces to implement `structured_output()` for `OllamaLLM`. First, we'll prepare the Ollama message data object that contains our instruction `prompt` by invoking our utility `chat_message_to_ollama_message()`. Next, we invoke the `ollama.chat()` method, passing the Ollama message data object, as well as the JSON Schema associated with `mdl` to specify our desired output structure. The output of `ollama.chat()` is another Ollama data type, from which we must extract the JSON data needed to build the instance of `mdl` that we return as the final output of `structured_output()`. Figure 3.14 illustrates this entire process.

Figure 3.14 Integrating with Ollama's structured output interface, which is facilitated by their chat interface. A structured_output() invocation of OllamaLLM invokes the ollama.chat() but not before creating the Ollama message data object. The returned result is an ollama.ChatResponse from which we extract the JSON payload corresponding to mdl that we then use to validate and create the final structured output object.


OllamaLLM
.model
._client
Prompt string
prompt
Pass prompt and desired structured output model
.structured_output(prompt, mdl)
outputs
An instance of mdl
mdl
A subclass of pydantic.BaseModel
mdl
create Ollama message data object with prompt
chat_message_to_ollama_message()
mdl.model_json_schema()
.chat()
.ChatResponse
mdl.validate()
query LLM
use Ollama's chat interface
extract JSON payload and create instance of mdl
Ollama server

The implementation of `structured_output()` for `OllamaLLM` is provided in the following listing.

Listing 3.8 Implementing OllamaLLM: structured_output()

```
# llm_agents_from_scratch/llms/ollama/llm.py
from typing import Any
from llm_agents_from_scratch.base.llm import (
    BaseLLM,
    StructuredOutputType,
)
from llm_agents_from_scratch.llms.utils import (
    chat_message_to_ollama_message,
)
from llm_agents_from_scratch.data_structures import (
    ChatMessage,
)

class OllamaLLM(BaseLLM):
    """Ollama LLM class."""
    …  #A

    async def structured_output(
        self,
        prompt: str,
        mdl: type[StructuredOutputType],
        **kwargs: Any,
    ) -> StructuredOutputType:
        """Structured output interface implementation for Ollama
LLM.
        …  #B
        """
        o_messages = [
            chat_message_to_ollama_message(
                ChatMessage(role="user", content=prompt),
            ),
        ]
        result = await self._client.chat(  #C
            model=self.model,
            messages=o_messages,
            format=mdl.model_json_schema(),  #D
        )
        return mdl.model_validate_json(result.message.content)
#E
```

note

For `structured_output()`, we could have also built the Ollama message data type directly rather than using our utility method `chat_message_to_ollama_message()`. I've elected to use it here to gradually introduce this concept, which we'll use again when implementing `chat()` and `continue_chat_with_tool_results()`.

To demonstrate the usage of `structured_output()`, we'll again prompt the `qwen2.5:3b` LLM to tell us a joke. This time, however, we'll have it output the joke in the format of the `Joke` class we defined earlier, which I show again here for convenience.

```
# Included in examples/ch03.ipynb  #A
import asyncio
from pydantic import BaseModel
from typing import Literal
from llm_agents_from_scratch.llms.ollama import OllamaLLM

class Joke(BaseModel):
    """A structured output model for Jokes."""

    subject: Literal["math", "physics", "biology"]
       joke: str


async def main():
    llm = OllamaLLM(model="qwen2.5:3b")
    prompt = ("Tell me a joke.")
    joke = await llm.structured_output(prompt=prompt, mdl=Joke)
#B
    print(joke.__class__.__name__)
    print(joke)

asyncio.run(main())
```

The first print statement outputs the class name of the returned joke, while the second print statement outputs the string representation of the returned `Joke` object. The following code snippet shows what the output of these two print statements should resemble.

```
Joke
subject='math', joke='Why did the math book look so sad? Because
it had lots of problems.'
```

Great! We have implemented two of the four interaction modes for `OllamaLLM`. Let's wrap up this section with our implementations of `chat()` and its convenient extension `continue_chat_with_tool_results()`. We've already encountered Ollama's chat interface, `ollama.chat()`, through our previous implementation of `structured_output()`. Naturally, we'll use `ollama.chat()` again to integrate with the chat interfaces of our framework.

note

We'll hold off on demonstrations of `chat()` and `continue_chat_with_tool_results()` until the next section, where we'll see them working together in a complete end-to-end demonstration of a tool-calling process.

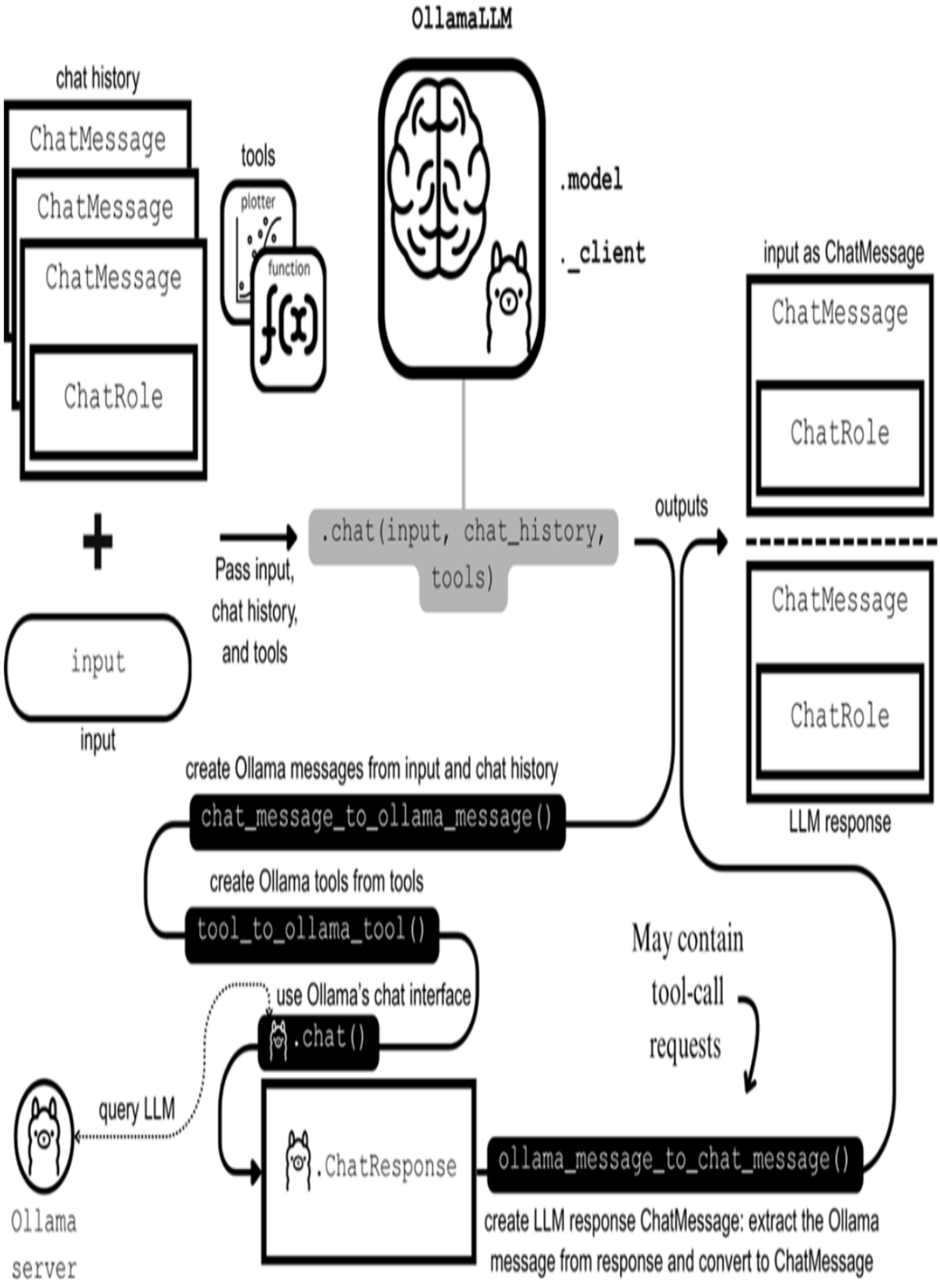
To build out this Ollama chat integration, we'll need to delve deeper into the inner workings of `ollama.chat()`. Specifically, `ollama.chat()` takes in an input list of Ollama message objects as well as an optional list of Ollama tool objects. After querying the LLM service, `ollama.chat()` returns an `ollama.ChatResponse` object from which we can extract the LLM's response message.

As mentioned earlier, building an integration with other LLM frameworks involves converting between the data types of the two frameworks. For this, we'll use our utility conversion functions.

We saw from figures 3.5 and 3.7 that `chat()` takes in three parameters: `input`, `chat_history`, and `tools`. We'll need to apply `chat_message_to_ollama_message()` to derive Ollama message objects from `input` and `chat_history`, and `tool_to_ollama_tool()` to derive Ollama tool objects from `tools`.

With our Ollama messages and tools prepared, we can invoke `ollama.chat()` and extract the LLM's response message from the invocation result. Since this message is an Ollama message data type, we'll need to apply the `ollama_message_to_chat_message()` to convert the LLM's response to a `ChatMessage`. We return this `ChatMessage` object along with another one that represents the user's input as the final output to the `chat()` method. Figure 3.15 shows this entire process.

Figure 3.15 Integrating with Ollama's chat interface to implement chat(). The user's input and chat history are converted to Ollama messages, and tools are converted to Ollama tools. Next, ollama.chat() is invoked and its response is then converted back to a ChatMessage. This ChatMessage is returned along with another one derived from the user's input.


OllamaLLM
chat history
ChatMessage
ChatMessage
ChatMessage
ChatRole
tools
plotter
function
.model
._client
input as ChatMessage
ChatMessage
ChatRole
outputs
.chat(input, chat_history, tools)
Pass input, chat history, and tools
input
input
ChatMessage
ChatRole
LLM response
create Ollama messages from input and chat history
chat_message_to_ollama_message()
create Ollama tools from tools
tool_to_ollama_tool()
May contain tool-call requests
use Ollama's chat interface
.chat()
query LLM
.ChatResponse
ollama_message_to_chat_message()
Ollama server
create LLM response ChatMessage: extract the Ollama message from response and convert to ChatMessage

The following code provides the implementation of `chat()`.

Listing 3.9 Implementing OllamaLLM: chat()

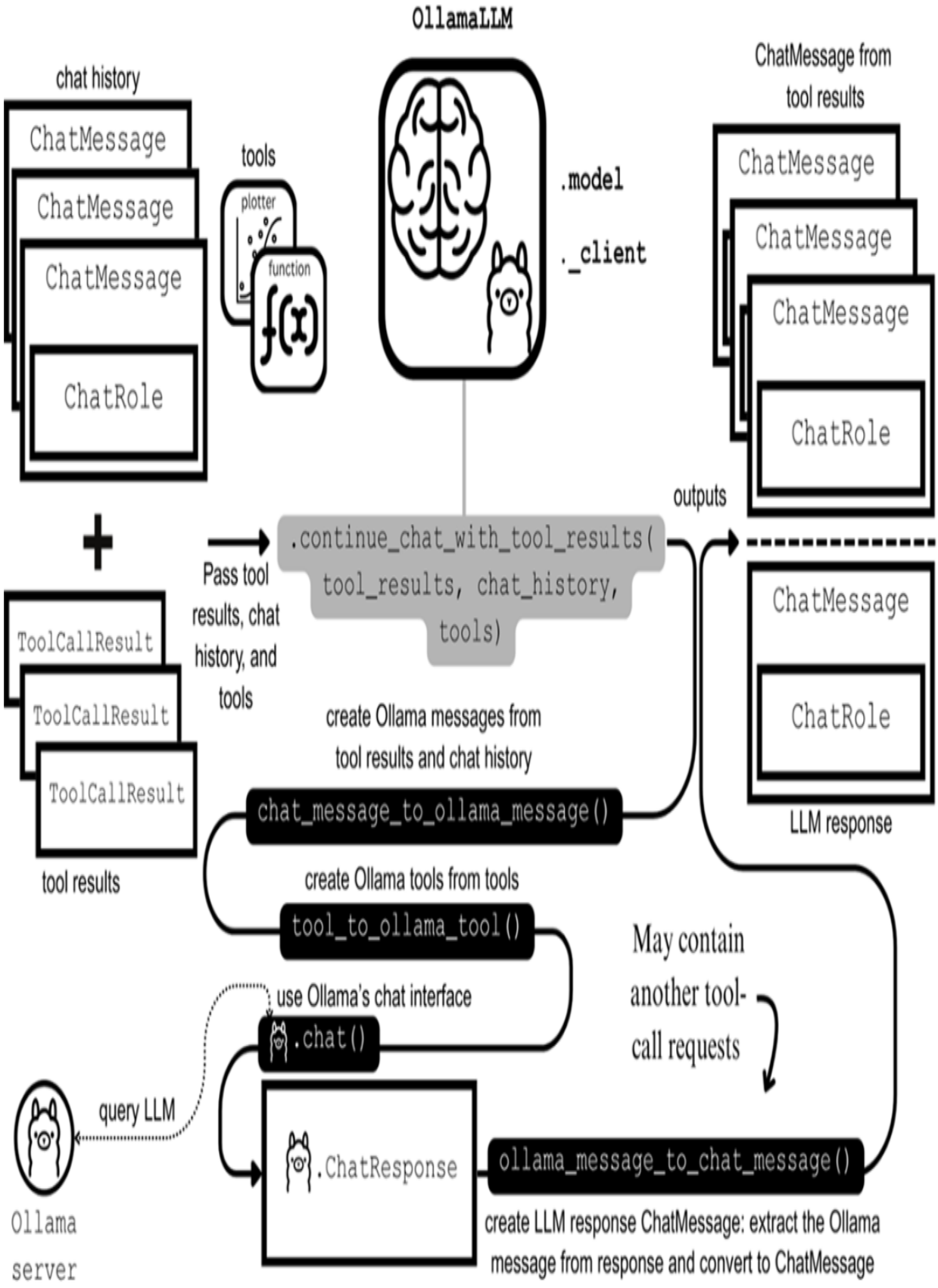
```
# llm_agents_from_scratch/llms/ollama/llm.py
…  #A
from llm_agents_from_scratch.llms.utils import (  #B
    chat_message_to_ollama_message,
    ollama_message_to_chat_message,
    tool_to_ollama_tool,
)

class OllamaLLM(BaseLLM):
    """Ollama LLM class."""
    …  #C

    async def chat(
        self,
        input: str,
        chat_history: list[ChatMessage] | None = None,
        tools: list[BaseTool | AsyncBaseTool] | None = None,
        **kwargs: Any,
    ) -> tuple[ChatMessage, ChatMessage]:
        …  #D
        # prepare messages  #E
        o_messages = (
            [chat_message_to_ollama_message(cm) for cm in
chat_history]
            if chat_history
            else []
        )
        user_message = ChatMessage(role="user", content=input)
        o_messages.append(
            chat_message_to_ollama_message(
                user_message,
            ),
        )

        # prepare tools  #F
        o_tools = (
            [tool_to_ollama_tool(t) for t in tools]
            if tools else None
        )

        result = await self._client.chat(  #G
            model=self.model,
            messages=o_messages,
```

```
        tools=o_tools,
    )

    return (
        user_message,
        ollama_message_to_chat_message(result.message)  #H
    )
```

Let's now implement the final abstract method, `continue_chat_with_tool_results()`. As with the implementation for `chat()`, much of the work we need to do here involves converting `ChatMessage` objects from our framework to the Ollama message data type equivalents. The main convenience this method offers, however, is that users can pass in `ToolCallResult` objects directly—they do not have to deal with converting these to `ChatMessage` objects at all.

To convert a `ToolCallResult` object to a `ChatMessage` object, we can use the constructor method, `from_tool_call_result()`, which we provided back in listing 3.2. From here, the process is similar to that of the previously implemented `chat()` method. We apply `chat_message_to_ollama_message()` and `tool_to_ollama_tool()` to obtain Ollama messages and tools. Next, we invoke `ollama.chat()` to have the LLM synthesize the tool results and generate a response. The result of `ollama.chat()` is converted back to a `ChatMessage` with `ollama_message_to_chat_message()`. We return this along with the list of `ChatMessage` objects constructed from the supplied `ToolCallResult` objects. Figure 3.16 illustrates this integration.

Figure 3.16 Integrating with Ollama's chat interface to implement continue_chat_with_tool_results(). The tool-call results and chat history are converted to Ollama messages, and tools are converted to Ollama tools. Next, ollama.chat() is invoked and its response is then converted back to a ChatMessage. This ChatMessage is returned along with the list of ChatMessage objects derived from the tool-call results.


OllamaLLM
chat history
ChatMessage
ChatMessage
ChatMessage
ChatRole
tools
plotter
function
.model
._client
ChatMessage from tool results
ChatMessage
ChatMessage
ChatMessage
ChatRole
outputs
.continue_chat_with_tool_results(tool_results, chat_history, tools)
Pass tool results, chat history, and tools
ChatMessage
ChatRole
LLM response
ToolCallResult
ToolCallResult
ToolCallResult
tool results
create Ollama messages from tool results and chat history
chat_message_to_ollama_message()
create Ollama tools from tools
tool_to_ollama_tool()
May contain another tool-call requests
use Ollama's chat interface
.chat()
query LLM
.ChatResponse
ollama_message_to_chat_message()
Ollama server
create LLM response ChatMessage: extract the Ollama message from response and convert to ChatMessage

The following code implements `continue_chat_with_tool_results()`.

Listing 3.10 Implementing OllamaLLM: continue_chat_with_tool_results()

```
# llm_agents_from_scratch/llms/ollama/llm.py
…  #A
class OllamaLLM(BaseLLM):
    """Ollama LLM class."""
    …  #B

    async def continue_chat_with_tool_results(
        self,
        tool_call_results: Sequence[ToolCallResult],
        chat_history: Sequence[ChatMessage],
        tools: Sequence[Tool] | None = None,
        **kwargs: Any,
    ) -> tuple[list[ChatMessage], ChatMessage]:
        …  #C
        # augment chat messages and convert to Ollama messages
        tool_messages = [
            ChatMessage.from_tool_call_result(tc)  #D
            for tc in tool_call_results
        ]
        o_messages = [  #E
            chat_message_to_ollama_message(cm)
            for cm in chat_history
        ] + [
            chat_message_to_ollama_message(tm)
            for tm in tool_messages
        ]

        # prepare tools
        o_tools = (
            [tool_to_ollama_tool(t) for t in tools]
            if tools else None
        )

        # send chat request
        o_result = await self._client.chat(  #F
            model=self.model,
            messages=o_messages,
        )

        return (
            tool_messages,  #G
            ollama_message_to_chat_message(o_result.message)  #H
        )
```

Phew. It was really no small feat to implement all four required methods for `OllamaLLM`. The payoff for all this work, though, is quite significant. We can now use any of the open-source LLMs available through Ollama in our framework! To wrap up this chapter, we'll revisit our Hailstone tool from the previous chapter to demonstrate how our `OllamaLLM` can perform a tool call with it.

Exercise 3.1 Structured outputs with an OllamaLLM

Create a structured data model for a poem that also includes the poem text and its type (e.g., sonnet, haiku, etc.). Use an `OllamaLLM` instance to generate a structured poem using your poem data model.

3.2.2 Hailstone tool call with OllamaLLM

Let's quickly revisit the Hailstone tool that we created in the previous chapter and demonstrate how we can use it to perform an entire tool-calling process with our new `OllamaLLM`. To do so, we'll make use of the `chat()` and `continue_chat_with_tool_results()` methods.

We'll use the same `qwen2.5:3b` model for our `OllamaLLM` and will supply it with the `hailstone_tool` that we created in the previous chapter.

note

If you are coding these examples for yourself and not using the provided Jupyter notebook, then be sure to bring the `hailstone_step_func()` defined in the previous chapter into the necessary scope e.g. in your `.py` file if coding with Python scripts.

To initiate a tool-calling process, we invoke `chat()` by supplying an input string that will elicit a Hailstone tool-call request, as shown in the following code.

```
# Included in examples/ch03.ipynb  #A
import asyncio
from llm_agents_from_scratch.llms.ollama import OllamaLLM
from llm_agents_from_scratch.data_structures.llm import
ChatMessage
```

```
from llm_agents_from_scratch.tools import SimpleFunctionTool

llm = OllamaLLM(model="qwen2.5:3b")  #B
hailstone_tool = SimpleFunctionTool(hailstone_step_func)  #C

async def main():
    user_input = (
        "What is the result of taking the next step of the "  #D
        "Hailstone sequence on the number 3?\n\n"
        "Be very succinct in your response."
    )
    return await llm.chat(  #E
        user_input,
        tools=[hailstone_tool],  #F
    )

user_msg, response_msg = asyncio.run(main())
print(response_msg.tool_calls)
```

The print statement should return a single tool call of Hailstone, as shown below. If it does, then congratulations, your `OllamaLLM` has successfully requested a tool call for our Hailstone tool.

```
[ToolCall(id_='6f6d02ca-56db-4872-8c8f-3814e2ceeb19',
tool_name='hailstone', arguments={'x': 3})]
```

As we know from before, however, making the LLM elicit a tool call request is only the first step in performing tool calls with LLMs. We next need to run the tool with the arguments and then pass the `ToolCallResult` back to the LLM.

```
# Included in examples/ch03.ipynb  #A
tool_call = response_msg.tool_calls[0]
tool_call_result = hailstone_tool(tool_call)  #B
print(tool_call_result)
```

As we saw in the last chapter, performing a tool call with a `BaseTool` returns a `ToolCallResult`. Thus printing the `tool_call_result` variable in the above snippet should resemble something like the following:

```
tool_call_id='6f6d02ca-56db-4872-8c8f-3814e2ceeb19',
content='10', error=False
```

With our `ToolCallResult` in hand, we can now finally submit this back to our LLM for synthesis and response using the `continue_chat_with_tool_results()` method. As you know, this method returns `ChatMessage` objects created from the tool results and the LLM's response.

```
# Included in examples/ch03.ipynb
async def main():
    return await llm.continue_chat_with_tool_results(
        tool_call_results=[tool_call_result],
        chat_history=[user_msg, response_msg],
    )

tools_msg, final_response = asyncio.run(main())
print(final_response)
```

The print statement should output the string representation of a `ChatMessage` object that looks something like the following:

```
role=<ChatRole.ASSISTANT: 'assistant'> content='The result of
taking the next step in the Hailstone sequence on the number 3
is 10.' tool_calls=None
```

We've done it! It took some time, but we have finally performed a complete tool-calling process using our Hailstone tool and our newly implemented `OllamaLLM` class. We've truly reached a significant milestone in our journey. In fact, we're now in the position to develop our first LLM agent from scratch. In the next chapter, we'll do just that by implementing the `LLMAgent` class.

Exercise 3.2 Performing an asynchronous tool call with OllamaLLM

Create an instance of `OllamaLLM` and perform a tool call using the asynchronous version of the Hailstone tool that you developed in Exercise 2.2.

3.3 Summary

- APIs provided by LLM providers typically support two modes of interaction with LLMs: one for text completion and another for

conversational dialogue.

- Our `BaseLLM` class supports text completion via the `complete()` method, and the conversational mode of interaction through our `chat()` method.
- One important use case for LLMs is structured output, where we instruct an LLM to produce an output that conforms to a specified data format, often described in JSON.
- Structured outputs with our `BaseLLM` class, can be performed via the `structured_output()` method.
- Integrations into LLM providers and their companion Python SDKs can be added to our framework by subclassing `BaseLLM`.
- The `OllamaLLM` integration implemented in this chapter allows for the use of any open-source LLMs supported by Ollama in our framework.

Appendix C. Implementing The PydanticFunctionTool

In Chapter 2, we introduced the PydanticFunctionTool and its usage pattern with our framework, as an alternative to our from-scratch SimpleFunctionTool. The main benefits of the PydanticFunctionTool are more robust JSON Schema generation for the associated function's parameters, as well as more powerful validation through the `pydantic` library. In this appendix, we provide a walkthrough on the full implementation of the `PydanticFunctionTool` as well as its asynchronous version, `AsyncPydanticFunctionTool`.

C.1 Implementing PydanticFunctionTool

We now will build the `PydanticFunctionTool` wrapper class that will enable the usage pattern that we just showed. As we saw then, we wrap a function which takes in a single parameter `params` that is `pydantic.BaseModel`. For convenience and to conform to our typing practices that we've begun to establish in our framework, let's create a designated type for these kinds of functions.

We'll call such function types `PydanticFunction` since they package their parameters in a `pydantic.BaseModel` type as we saw in `HailstoneParams`. The next listing provides its definition.

Listing C.1 Implementing the PydanticFunction Protocol

```
# llm_agents_from_scratch/tools/pydantic_function.py
from typing import Any, Protocol
from pydantic import BaseModel


class PydanticFunction(Protocol):
    """PydanticFunction Protocol."""
```

```
    __name__: str
    __doc__: str | None

    def __call__(
        self,
        params: BaseModel,
        *args: Any,
        **kwargs: Any
    ) -> Any:
        """Callable interface.

        Args:
            params (BaseModel): The function's params as a
                ~pydantic.BaseModel.
            *args (Any): Additional positional arguments.
            **kwargs (Any): Additional keyword arguments.

        Returns:
            Any: The result of the function call.
        """
        ...  #A
```

The listing above shows that we have chosen to implement the `PydanticFunction` as a subclass of `typing.Protocol`, which are meant for defining interfaces that can be typed-checked at runtime. The lone method that we requires with this interface is the special `__call__` method, which importantly takes in a parameter, `params`, that is of type `pydantic.BaseModel`. As an interface method, we need only supply elipsis (i.e., …), which in this context indicates that the actual function provides the implementation.

The other attributes we require in this interface are special Python attributes `__name__` and `__doc__`, which are defined by default for Python functions. In other words, any function that has a signature involving a single variable named `params` of type `pydantic.BaseModel`, satisfies the `PydanticFunction` protocol—this is precisely what we need.

Another added benefit of defining our `PydanticFunction` as a `typing.Protocol` is that modern Integrated Development Environments (IDEs), such as VSCode, support typing and provide helpful type hints.

Now that we have defined a special type for our functions to be wrapped by `PydanticFunctionTool`, let's move onto the next order of business. To make our implementation more robust, we should ensure that any function that we wish to wrap with this class is indeed a `PydanticFunction`. We will handle this by implementing specific validation logic next.

Listing C.2 Validating functions by signature inspection

```
# llm_agents_from_scratch/tools/pydantic_function.py
import inspect
from typing import Any, Awaitable, Callable, Protocol,
get_type_hints
from pydantic import BaseModel


def _validate_pydantic_function(func: Callable) ->
type[BaseModel]:
    """Validates func as a proper `PydanticFunction`.

    Args:
        func (Callable): The function to validate as
`PydanticFunction`.

    Raises:
        RuntimeError: If validation of `func` fails.
    """
    sig = inspect.signature(func)
    type_hints = get_type_hints(func)

    if "params" not in sig.parameters:  #A
        raise RuntimeError(
            "Validation of `func` failed: Missing `params`
argument.",
        )

    if annotation := type_hints.get("params"):
        if not issubclass(annotation, BaseModel):  #B
            msg = (
                f"Validation of `func` failed: {annotation} is
not"
                " a subclass of `~pydantic.BaseModel`."
            )
            raise RuntimeError(msg)
    else:
        msg = (
```

```
            "Validation of `func` failed: `params` argument
must have "
            "type annotation."
        )
        raise RuntimeError(msg)

    return annotation  #C
```

Let's walkthrough our validation logic step-by-step. The first bit of code should hopefully look familiar, as we're again needing to perform introspection on the function signature, as we did when we built logic to transform function signatures to JSON Schemas from scratch for `SimpleFunctionTool`. As we did then, we rely on the combination of `inspect.signature` and `typing.get_type_hints` here to extract the parameters and their annotated types from the supplied function's signature.

We then check if a parameter with the specific name "params" exists in the extracted parameters. If it does, then we next check that its annotated type is a `pydantic.BaseModel`. If either of these checks fail, then we raise a `RunTimeError` with a message indicating as to why the validation failed.

If both checks pass, then we have a valid `PydanticFunction`, and we return the type or subclass of `pydantic.BaseModel` with which `params` is annotated. Let's quickly demonstrate this validation with our `hailstone_pydantic_fn`.

```
from llm_agents_from_scratch.tools.pydantic_function import (
    _validate_pydantic_function,
)

print(_validate_pydantic_function(hailstone_pydantic_fn))
```

The above print statement should return an output similar to the following.

```
__main__.HailstoneParams
```

We are now in a position to start implementing our `PydanticFunctionTool`, which, like `SimpleFunctionTool`, inherits our `BaseTool` class. As we have already seen how to create a subclass of `BaseTool`, we'll present the entire implementation in the listing below and then discuss it, rather than implementing methods incrementally as we did before.

Listing C.3 Implementing PydanticFunctionTool

```
#llm_agents_from_scratch/tools/pydantic_function.py
import inspect
from typing import Any, Awaitable, Callable, Protocol,
get_type_hints
from pydantic import BaseModel
from llm_agents_from_scratch.base.tool import BaseTool
from llm_agents_from_scratch.data_structures import (
    ToolCall,
    ToolCallResult,
)

…  #A

class PydanticFunctionTool(BaseTool):
    """Pydantic function calling tool.

    Turn a Python function that takes in a ~pydantic.BaseModel
params
    Object into a tool for an LLM.

    Attributes:
        func: PydanticFunction to represent as a tool.
        params_mdl: The params BaseModel.
        desc: Description of the PydanticFunction.
    """

    def __init__(
        self,
        func: PydanticFunction,
        desc: str | None = None,
    ):
        """Initialize a PydanticFunctionTool.

        Args:
            func (PydanticFunction): The Pydantic function to
expose as a
                tool to the LLM.
            desc (str | None, optional): Description of the
function.
                Defaults to None.
        """
        self.func = func
        self.desc = desc or func.__doc__ or f"Tool for
{func.__name__}"
        self.params_mdl = _validate_pydantic_function(func)  #B
```

```
    @property
    def name(self) -> str:
        """Name of function tool."""
        return self.func.__name__

    @property
    def description(self) -> str:
        """Description of what this function tool does."""
        return self.desc

    @property
    def parameters_json_schema(self) -> dict[str, Any]:
        """JSON schema for tool parameters."""
        return self.params_mdl.model_json_schema()  #C

    def __call__(
        self,
        tool_call: ToolCall,
        *args: Any,
        **kwargs: Any,
    ) -> ToolCallResult:
        """Execute the function tool with a ToolCall.

        Args:
            tool_call (ToolCall): The ToolCall to execute.
            *args (Any): Additional positional arguments.
            **kwargs (Any): Additional keyword arguments.

        Returns:
            ToolCallResult: The result of the tool call
execution.
        """
        try:
            params =
self.params_mdl.model_validate(tool_call.arguments)  #D
            # execute the function
            res = self.func(params)  #E
            content = str(res)
            error = False
        except Exception as e:
            content = f"Failed to execute function call: {e}"
            error = True

        return ToolCallResult(
            tool_call=tool_call,
            content=content,
```

```
            error=error,
        )
```

Beginning with the `__init__` method, we can see that it has the exact same logic as `SimpleFunctionTool.__init__`, however, with the added statement for validation and automatic extraction of the params model. As we discussed, `_validate_pydantic_function` returns the type (or class) of the function's `params` argument. This can be seen as a bit of added convenience, as we don't require the user to supply this information, but rather extract it from the supplied `PydanticFunction` automatically.

Next, the implementations for the property attributes `name` and `desc` are the same as those for `SimpleFunctionTool`. Our current implementation of `parameters_json_schema`, however, is different from that of `SimpleFunctionTool`. Here, we simply call the method `params_mdl.model_json_schema`, which is a class method that returns the JSON Schema representation of `params_mdl` (a type of `pydantic.BaseModel`). Unlike `SimpleFunctionTool`, where we relied on our from-scratch code to convert function signatures to their JSON Schema representations, we now utilize the `pydantic` library to build this JSON Schema from the `params_mdl` for us.

Finally, our implementation of `__call__` also involves a delegation to the wrapped function `func`. However, since `func` is `PydanticFunction`, we need to pass it an instance of the `params_mdl` type, which we must build using the arguments supplied by the LLM in its tool call request. Again, here we leverage the pydantic library but this time for its powerful JSON validation. Specifically, we call the factory method params_mdl.model_validate, which takes in a Python dictionary of JSON data that is first validated against JSON Schema for `params_mdl`. If validation is successful, then the input data gets used to initialize an instance of `params_mdl`, which we store in the `params` variable and pass as input to `func`. The rest of our implementation for `__call__` is similar to that of `SimpleFunctionTool`, including our approach for handling any errors that may be encountered during the validation of tool parameters or the delegation to the wrapped function.

The `hailstone_pydantic_fn_tool` that we defined at the start of this section should work as intended with what we just developed.

```
print(hailstone_pydantic_fn_tool.description)
print(hailstone_pydantic_fn_tool.name)
print(hailstone_pydantic_fn_tool.parameters_json_schema)
```

These statements should print the following results.

```
hailstone_pydantic_fn
Perform a single step of the Hailstone sequence.
{'properties': {'x': {'title': 'X', 'type': 'integer'}},
'required': ['x'], 'title': 'HailstoneParams', 'type':
'object'}
```

We can also use `hailstone_pydantic_fn_tool` with the same tool call request that we had for our quick demonstration of `hailstone_tool` (a `SimpleFunctionTool`) in the previous section.

```
from llm_agents_from_scratch.data_structures import ToolCall

tool_call = ToolCall(
    tool_name="hailstone_pydantic_fn_tool",
    arguments={"x": 3}
)

res = hailstone_pydantic_fn_tool(tool_call)
print(res)
```

The print statement should print output as follows.

```
tool_call=ToolCall(tool_name='hailstone_pydantic_fn',
arguments={'x': 3}) content='10' error=False
```

This wraps up our implementation of `PydanticFunctionTool`, an alternative approach to building tools from Python functions using our LLM agent framework. Like `SimpleFunctionTool`, however, this wrapper class only works with synchronous functions. And, as we did then, we'll now need to quickly add an async version of this wrapper class to work with asynchronous functions.

C.2 The AsyncPydanticFunctionTool

Let's quickly add an async version of `PydanticFunctionTool` that works with asynchronous functions much like `AsyncSimpleFunctionTool` does. We will move rather quickly here since the implementation is very similar to the async version, with only minor differences that are similar to the differences observed between `SimpleFunctionTool` and `AsyncSimpleFunctionTool`.

First, we'll require an async version of our PydanticFunction protocol. We'll call this function protocol `AsyncPydanticFunction` and present its implementation in the next listing.

Listing C.4 Implementing the AsyncPydanticFunction Protocol

```
# llm_agents_from_scratch/tools/pydantic_function.py
from typing import Any, Protocol
from pydantic import BaseModel


class AsyncPydanticFunction(Protocol):
    """Asynchronous PydanticFunction Protocol."""

    __name__: str
    __doc__: str | None

    async def __call__(  #A
        self,
        params: BaseModel,
        *args: Any,
        **kwargs: Any,
    ) -> Awaitable[Any]:  #B
        """Asynchronous callable interface.

        Args:
            params (BaseModel): The function's params as a
                ~pydantic.BaseModel.
            *args (Any): Additional positional arguments.
            **kwargs (Any): Additional keyword arguments.

        Returns:
            Awaitable[Any]: The result of the function call.
        """
        ...
```

The only difference between this `AsyncPydanticFunction` and `PydanticFunction` from before is that the async protocol stipulates that the function's `__call__` method be an async function that returns an `~typing.Awaitable` type.

With this async function type defined, we can now present the complete implementation of `AsyncPydanticFunctionTool`.

Listing C.5 Implementing AsyncPydanticFunctionTool

```
# llm_agents_from_scratch/tools/pydantic_function.py
import inspect
from typing import Any, Awaitable, Callable, Protocol,
get_type_hints
from pydantic import BaseModel
from llm_agents_from_scratch.base.tool import AsyncBaseTool,
BaseTool
from llm_agents_from_scratch.data_structures import (
    ToolCall,
    ToolCallResult,
)


class AsyncPydanticFunctionTool(AsyncBaseTool):
    """Async Pydantic function calling tool.

    Turn an async Python function that takes in a
~pydantic.BaseModel
    Params object into a tool for an LLM.

    Attributes:
        func: AsyncPydanticFunction to represent as a tool.
        params_mdl: The params BaseModel.
        desc: Description of the PydanticFunction.
    """

    def __init__(
        self,
        func: AsyncPydanticFunction,
        desc: str | None = None,
    ):
        """Initialize an AsyncPydanticFunctionTool.

        Args:
            func (AsyncPydanticFunction): The async Pydantic
```

```
function to
                expose as a tool to the LLM.
            desc (str | None, optional): Description of the
function.
                Defaults to None.
        """
        self.func = func
        self.desc = desc or func.__doc__ or f"Tool for
{func.__name__}"
        self.params_mdl = _validate_pydantic_function(func)

    @property
    def name(self) -> str:
        """Name of function tool."""
        return self.func.__name__

    @property
    def description(self) -> str:
        """Description of what this function tool does."""
        return self.desc

    @property
    def parameters_json_schema(self) -> dict[str, Any]:
        """JSON schema for tool parameters."""
        return self.params_mdl.model_json_schema()

    async def __call__(
        self,
        tool_call: ToolCall,
        *args: Any,
        **kwargs: Any,
    ) -> ToolCallResult:
        """Execute the function tool with a ToolCall.

        Args:
            tool_call (ToolCall): The ToolCall to execute.
            *args (Any): Additional positional arguments.
            **kwargs (Any): Additional keyword arguments.

        Returns:
            ToolCallResult: The result of the tool call
execution.
        """
        try:
            params =
self.params_mdl.model_validate(tool_call.arguments)
            # execute the function
            res = await self.func(params)
```

```
        content = str(res)
        error = False
    except Exception as e:
        content = f"Failed to execute function call: {e}"
        error = True

    return ToolCallResult(
        tool_call=tool_call,
        content=content,
        error=error,
    )
```

The implementation of `AsyncPydanticFunctionTool` is very similar to that of `PydanticFunctionTool`, with the only differences being that here we inherit from `AsyncBaseTool` and that the `__call__` method is now async.

FROM SCRATCH

BUILD A

Multi-Agent System

With MCP and A2A

Val Andrei Fajardo

MEAP

MANNING