4

The AI Ecosystem: Putting the Pieces Together

Introduction

Whether you're a product manager, machine learning engineer, CEO, or even just someone who has the urge to build things, by the time you get to the part where you're actually designing an AI-enabled product or feature, you run into a question that everyone faces: *How in the world do I turn raw AI power into a usable, delightful experience?*

The past few chapters have focused on individual components of what makes most AI features great, including these:

- An understanding of the different types of LLMs (autoencoding versus autoregressive) and what kinds of tasks they excel at
- Seeing how closed- and open-source LLMs can work together in applications like semantic search
- Getting the most out of LLMs using structured prompt engineering and how that leads to more agnostic deployments of prompts and models

We have even hinted at the idea of starting to put these ideas together into comprehensive AI-enabled features—and that's exactly what this chapter is about. To that end, we will walk through two currently popular applications of LLMs for two reasons: because their popularity signals that many of you are considering building something similar, and because they offer evergreen techniques and considerations that future AI applications will come up against.

If the first section of this book had a moral that I hope you take away from reading it, it is this: The best AI applications do *not* simply rely on the raw power of an AI model, whether it is fine-tuned or not. Rather, it's the ecosystem of AI models and tools that make the application shine and persist for a long period of time.

The Ever-Shifting Performance of Closed-Source AI

Our last chapter on prompt engineering showed that by structuring prompts we can achieve the most consistent results and become more model agnostic. It then becomes easy to believe that simply prompting well and using a powerful model is enough to power your AI application—provided the cost projections work out in your favor (a theme throughout this book). To be frank, prompting well and setting up a test suite (more on that later in this chapter) can be enough for some smaller individual features of a larger application. I will go on record and state that a majority of the AI features I deploy for my own startups fall into the category of "prompt well and test often."

One of the main issues with solely relying on a model, especially closed-source ones from for-profit entities, is that the companies have complete control over how often they update their models using new data, new techniques, new architectures—really, new anything. For example, if you design a prompt for a model, say GPT-3.5 in January 2024, it may not transfer over to an updated version of that model, perhaps GPT-3.5 in May 2024. Let's take OpenAI's GPT 4 as a concrete example. OpenAI updates its models every few months so that the model can have more data attached to it. "gpt-3.5-turbo-1106" refers to the model released on November 6, whereas "gpt-3.5-turbo-0613" refers to the model released on June 13. Both versions are GPT 3.5 (ChatGPT), but they contain different model weights and therefore have different behaviors. In turn, they should be considered to be separate models.

Let's look at a concrete example of this behavior change from a paper from 2023 titled "How Is ChatGPT's Behavior Changing over Time?"[1] In this study, the authors took some prompts and tasks and put them to the test on four different models:

---

1. https://arxiv.org/abs/2307.09009

- GPT 3.5 from March 2023 (gpt-3.5-turbo-0314)
- GPT 3.5 from June 2023 (gpt-3.5-turbo-0613)
- GPT 4 from March 2023 (gpt-4-0314)
- GPT 4 from June 2023 (gpt-4-0613)

The idea was to see if simply asking the model to solve a task (often using a chain-of-thought prompt) would lead to changes in performance on different versions of both GPT 3.5 and GPT 4. The answer, as you probably guessed from the fact that I'm even bringing this up, is yes—yes, it did show changes in behavior. I'll point out one specific task as our primary example, but I encourage you to check out the paper and the full results. Figure 4.1 highlights the example of asking the four models whether a number is a prime number.

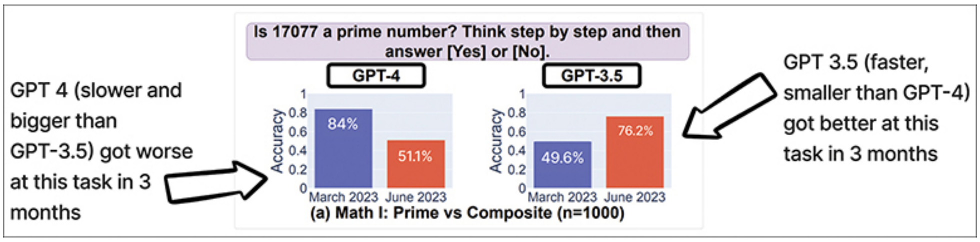

Figure 4.1 In just one of the tasks the Stanford/Berkeley team tested, both models showed a large delta in performance. Source: Chen, L., et al. "How Is ChatGPT's Behavior Changing over Time?" (2023). Retrieved from https://arxiv.org/abs/2307.09009.

Even with just a three-month gap, the GPT-4 model got much worse at this task whereas the GPT-3.5 model got better! This is not a reason to boycott OpenAI or its models by any means, but simply a consequence of frequent training for the purpose of trying to force the models to be good at as many things as possible for as many people as possible. Inevitably, there will be swings in downstream task-specific performance that affect the individual.

Deliberate and structured prompting with a decently sized testing suite can sometimes suffice for smaller AI features, but it is often not enough when we want to tackle the larger, more complex applications. One of the main reasons we see this delta in difficulty is that current LLM architectures excel much more at reasoning through a given context than they do at recalling encoded information from their parameters and potentially hallucinating in the process.

AI Reasoning versus Thinking

It might be a mildly controversial thing to say that current generative LLMs like Gemini, Claude 3, Llama 3, and GPT-4 are better at reasoning through a given context than they are at "thinking." I should be clear here: By "thinking," I am referring to an LLM's ability to recall encoded information on its own with no explicit context from the prompt. If you take a step back from individual LLM outputs, you might also notice that AI systems tend to have a "voice" or a "style" all their own, and that style can often be monotone and repetitive. Moreover, this “AI tone” can even be seen across models.

As an example, I put the exact same prompt into Google’s Gemini, Cohere’s Command R model, Anthropic’s Claude Sonnet, and OpenAI’s GPT-4. As **Figure 4.2** shows, I got strikingly similar responses, many of which draw directly from the input text. I asked each model to summarize the chapter you’re reading now using at most five sentences.

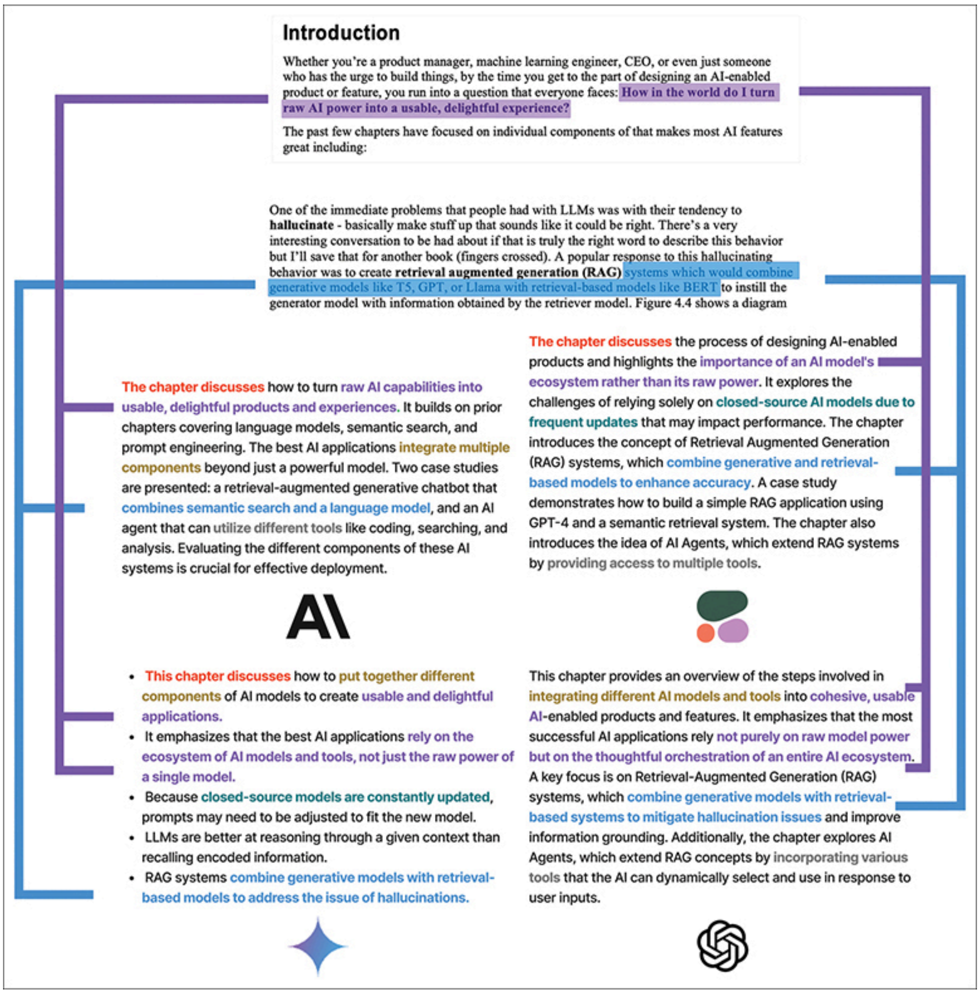

Figure 4.2 Asking Claude Sonnet (top left), Gemini (bottom left), Command R (top right), and GPT-4 (bottom right) to write a paragraph summarizing the chapter you are currently reading using the exact same prompt ("Could you summarize this book chapter for me please in 5 sentences?", followed by the text of this chapter) yielded similar results, all drawing from my original text—sometimes verbatim. They all essentially remix what I wrote rather than coming up with brand-new sentences. This isn't a bad thing; it just illustrates how most generative AI models, when given a context, will favor using the context directly rather than coming up with their own wording.

In **Chapter 3**, which introduced prompt engineering, we saw that the best way to entice a generative AI to be consistent and produce outputs in the style we want is to provide examples through few-shot learning and to force the AI to reason first through chain-of-thought prompting. **Figure 4.3** serves as a reminder that models like GPT-4 are more accurate when they have to reason through a problem first before answering (reasoning) than when they have to conjure up an answer on the spot.

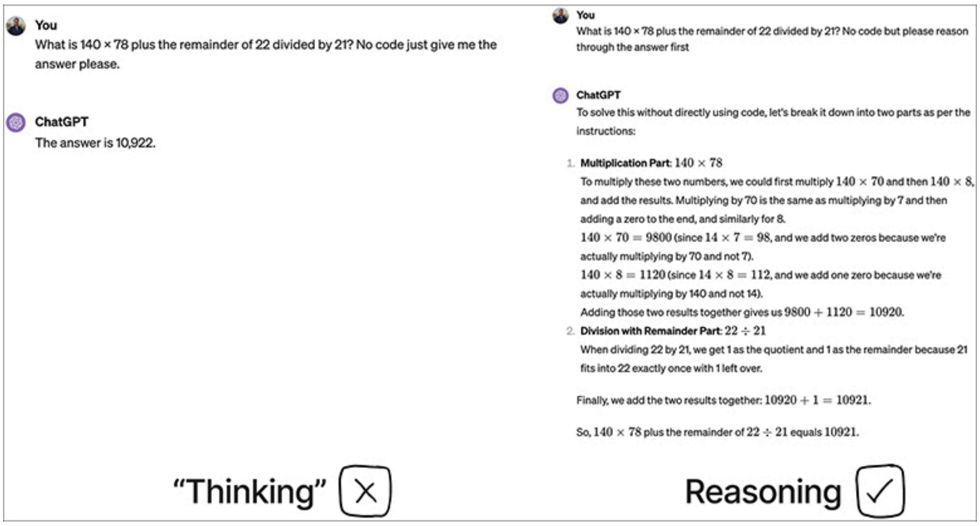

Figure 4.3 Invoking reasoning through chain-of-thought prompting leads to the correct answer of 10,921 at the cost of a deluge of output tokens (more $$).

In this chapter we will tackle two popular AI applications that build upon these prompting fundamentals. We will build prompts with chain of thought, few-shot learning, prefix notation, and more with the aim of creating usable and delightful applications. In our first example, we'll integrate our semantic search system from **Chapter 2** to build a retrieval augmented generative chatbot. In our second example, we'll go even further and build a full AI agent connected to home-grown tools.

Case Study 1: Retrieval Augmented Generation

One of the immediate problems that people had with LLMs was with their tendency to **hallucinate**—basically, make stuff up that sounds as if it could be right. There's a very interesting conversation to be had about whether that is truly the right word to describe this behavior, but I'll save that for another book (fingers crossed). A popular response to this hallucinating behavior was to create **retrieval augmented generation (RAG)** systems, which combined generative models like T5, GPT, and Llama with retrieval-based models like BERT to fill the generator model with information obtained by the retriever model. Figure 4.4 shows a diagram from the original 2020 paper, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks."[2]

---

2. https://arxiv.org/abs/2005.11401

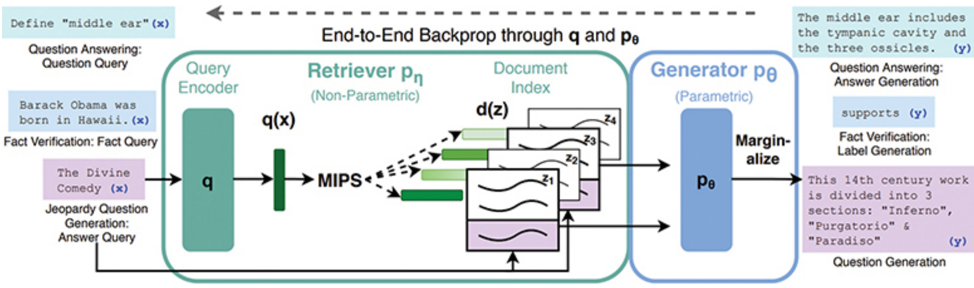

Figure 4.4 The original RAG paper includes more advanced training methods for fine-tuning RAG performance. Source: Lewis, P. et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." *Advances in Neural Information Processing Systems* 33 (2020): 9459-74. Retrieved from https://arxiv.org/abs/2005.11401.

We will build a very simple RAG application using GPT-4 and the semantic retrieval system we built in Chapter 2.

The Sum of Our Parts: The Retriever and the Generator

Our RAG system will have two parts:

- **A retriever:** Something to put ground-truth knowledge into a repository and an LLM to retrieve them given a query. Our semantic search API from Chapter 2 will be our retrieval operator.
- **A generator:** An LLM to reason through the user's query and the retrieved knowledge to provide an inline conversational response. This will be GPT-4.

Recall that one of our semantic search API endpoints was used to retrieve documents from our dataset given a natural query. All we need to do to get our RAG system off the ground is to complete four steps:

1. Design a system prompt for GPT-4 demonstrating the preferred conversational structure through few-shot learning and chain-of=thought prompting.
2. Kick off a query to our semantic search system when a human asks our bot a question. In Chapter 2, we did most of the hard work of chunking, vectorizing, and indexing. Now we get to simply use the system for what it was built for: accessing real-time context.
3. Inject any context we find from our DB directly into GPT-4's system prompt.
4. Let GPT-4 do its job and answer the question.

Figure 4.5 outlines these high-level steps.

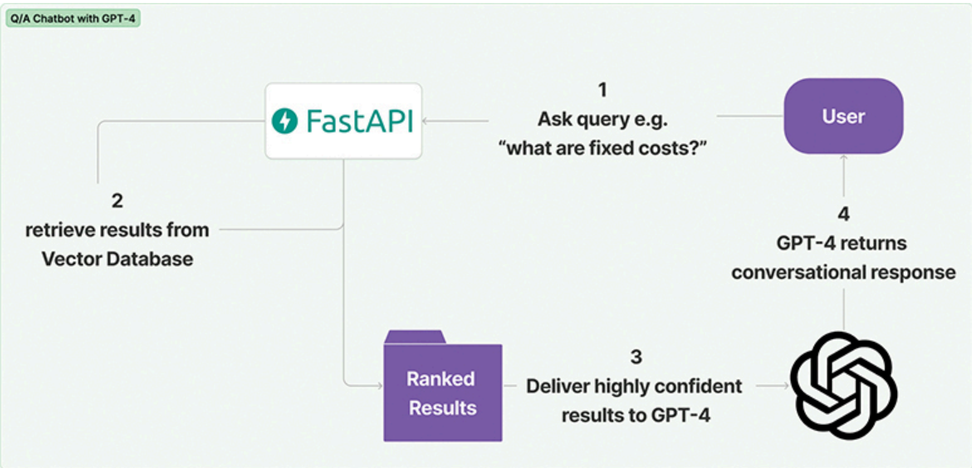

Figure 4.5 A 10,000-foot view of our retrieval-augmented generative chatbot that uses GPT-4 to provide a conversational interface in front of our semantic search API.

To dig into one step deeper, Figure 4.6 shows how this will work at the prompt level, step by step.

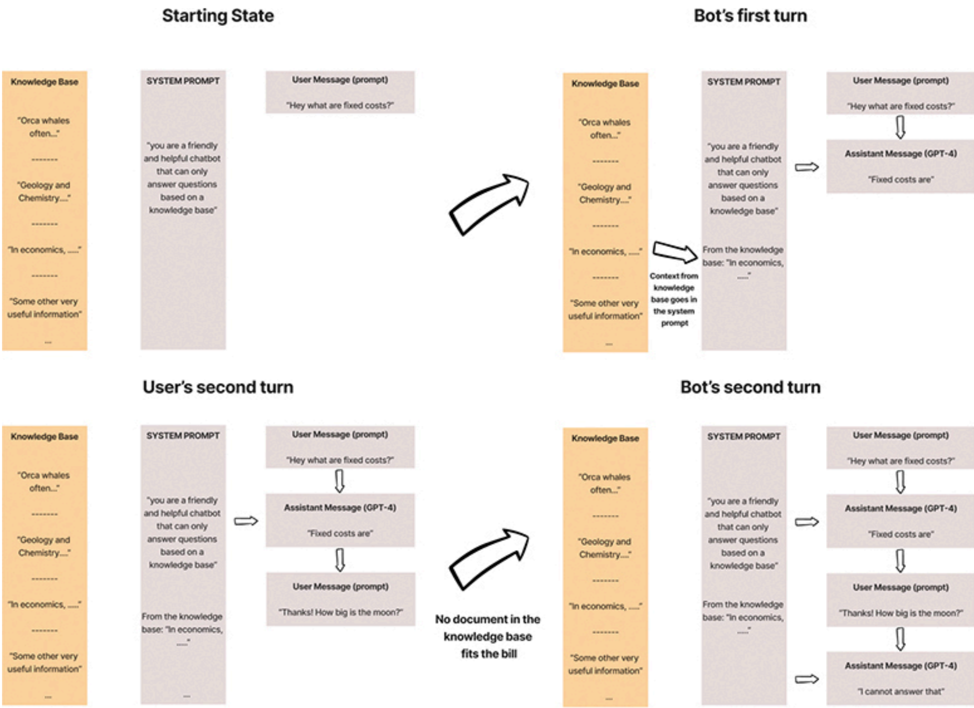

Figure 4.6 Starting from the top left and reading from left to right, these four states represent how our bot is architected. Every time a user says something that surfaces a confident document from our knowledge base, that document is inserted directly into the system prompt, where we tell GPT-4 to use only documents from our knowledge base.

Let's wrap all of this logic into a Python class, which will have a skeleton like in **Listing 4.1**.

Listing 4.1 A GPT-4 RAG bot

Click here to view code image

```
client = OpenAI(api_key=userdata.get('OPENAI_API_KEY'))

class ChatLLM(BaseModel):
    model: str = 'gpt-3.5-turbo'
    temperature: float = 0.0

    def generate(self, prompt: str, stop: List[str] = None):
        response = client.chat.completions.create(
            model=self.model,
            messages=[{"role": "user", "content": prompt}],
            temperature=self.temperature,
```

```
            stop=stop
        )
        return response.choices[0].message.content

FINAL_ANSWER_TOKEN = "Assistant Response:"
STOP = '[END]'
PROMPT_TEMPLATE = """Today is {today} and you can retrieve information from a
database. Respond to the user's input as best as you can.

Here is an example of the conversation format:

[START]
User Input: the input question you must answer
Context: retrieved context from the database
Context Score: a score from 0 to 1 of how strong a match the information is
Assistant Thought: This context has sufficient information to answer the quest
Assistant Response: your final answer to the original input question, which co
I don't have sufficient information to answer the question.
[END]
[START]
User Input: another input question you must answer
Context: more retrieved context from the database
Context Score : another score from 0 to 1 of how strong a match the informatio
Assistant Thought: This context does not have sufficient information to answer
question.
Assistant Response: your final answer to the second input question, which coul
I don't have sufficient information to answer the question.
[END]

Begin:

{running_convo}
"""

class RagBot(BaseModel):
    llm: ChatLLM
    prompt_template: str = PROMPT_TEMPLATE
    stop_pattern: List[str] = [STOP]
    user_inputs: List[str] = []
    ai_responses: List[str] = []
    contexts: List[Tuple[str, float]] = []

    def query_from_pinecone(self, query, top_k=1, include_metadata=True):
        return query_from_pinecone(query, top_k, include_metadata)

    @property
    def running_convo(self):
        convo = ''
        for index in range(len(self.user_inputs)):
            convo += f'[START]\nUser Input: {self.user_inputs[index]}\n'
            convo += f'Context: {self.contexts[index][0]}\nContext Score: {sel
contexts[index][1]}\n'
```

```
            if len(self.ai_responses) > index:
                convo += self.ai_responses[index]
                convo += '\n[END]\n'
        return convo.strip()

    def run(self, question: str):
        self.user_inputs.append(question)
        top_response = self.query_from_pinecone(question)[0]
        self.contexts.append(
            (top_response['metadata']['text'], top_response['score']))

        prompt = self.prompt_template.format(
                today = datetime.date.today(),
                running_convo=self.running_convo
        )
        generated = self.llm.generate(prompt, stop=self.stop_pattern)
        self.ai_responses.append(generated)
        return generated
```

Our bot has prefix notation, a chain of thought (by asking for the thought before the response), and an example of how a conversation should go (one-shot example). A full implementation of this code can be found in the book’s repository. **Figure 4.7** shows an example of a conversation we can have with it.

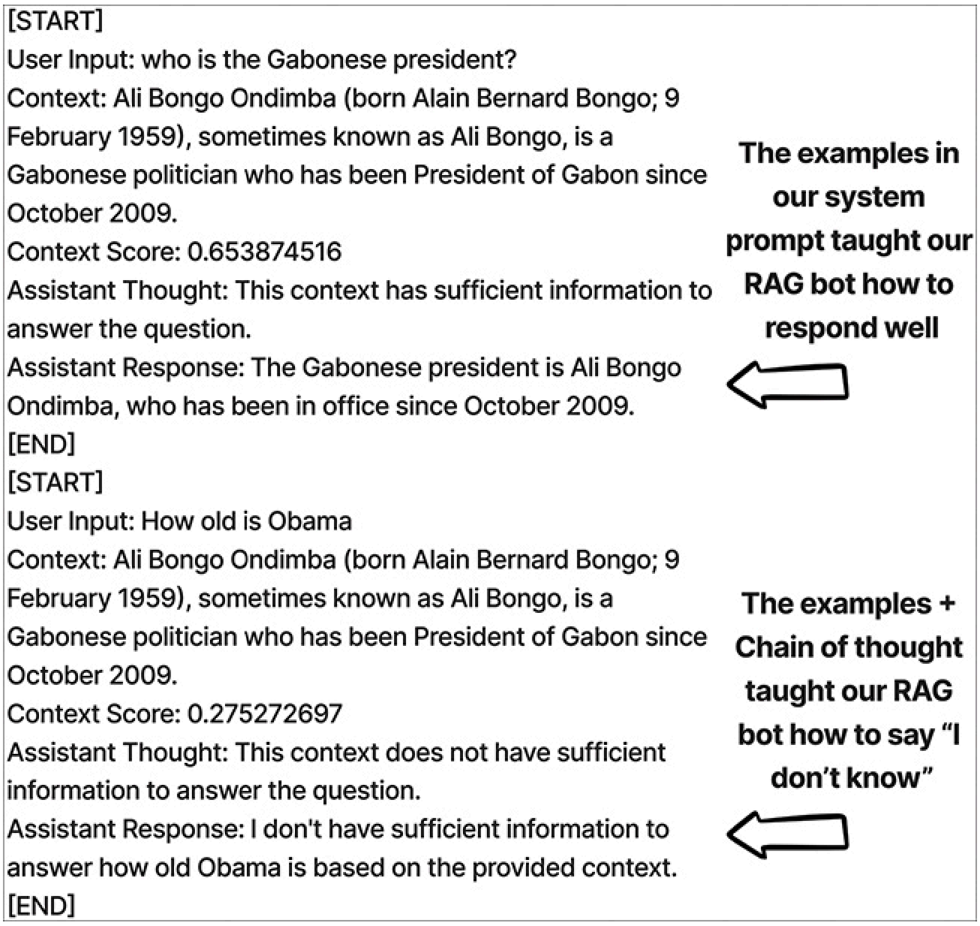
[START]
User Input: who is the Gabonese president?
Context: Ali Bongo Ondimba (born Alain Bernard Bongo; 9 February 1959), sometimes known as Ali Bongo, is a Gabonese politician who has been President of Gabon since October 2009.
Context Score: 0.653874516
Assistant Thought: This context has sufficient information to answer the question.
Assistant Response: The Gabonese president is Ali Bongo Ondimba, who has been in office since October 2009.
[END]
[START]
User Input: How old is Obama
Context: Ali Bongo Ondimba (born Alain Bernard Bongo; 9 February 1959), sometimes known as Ali Bongo, is a Gabonese politician who has been President of Gabon since October 2009.
Context Score: 0.275272697
Assistant Thought: This context does not have sufficient information to answer the question.
Assistant Response: I don't have sufficient information to answer how old Obama is based on the provided context.
[END]

Figure 4.7 Talking to our chatbot yields cohesive and conversational answers about the Gabonese president (note this was not true as of 2023, which highlights a data staleness issue). However, when I ask about Barack Obama's age (which is not in the database), the AI politely declines to answer even though that is general knowledge it would try to use otherwise.

As a fun side-test, I decided to try something out of the box. I built a new namespace in the vector database and chunked documents out of a PDF for a *Star Wars*–themed deck building game I like. I wanted to use the chatbot to ask basic questions about the game and let GPT-4 retrieve portions of the manual to answer my questions. The results can be seen in **Figure 4.8**.

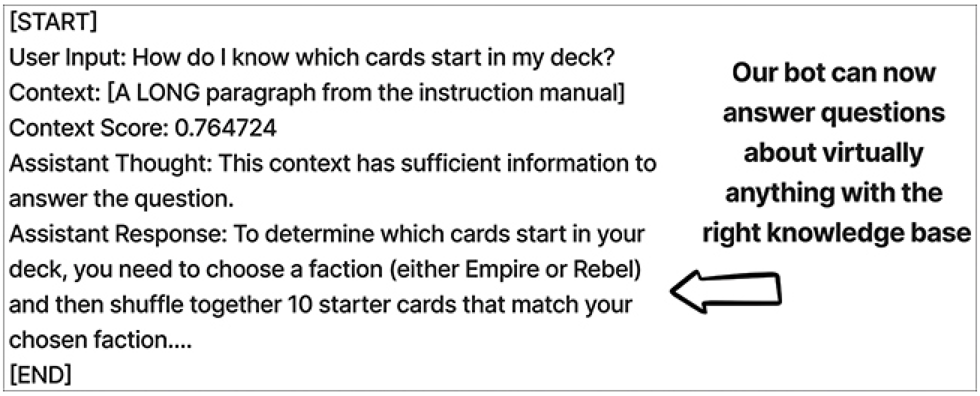

Figure 4.8 The same architecture and system prompt against a new knowledge base of a card game manual. Now I can ask questions about a board game I like and get on-demand help.

Not bad at all, if I may say so. Of course, these are singular examples of our bot and we should look at more rigorous testing of our RAG system.

Evaluating a RAG System

Evaluating a RAG system is really evaluating the two components separately:

- The retriever: How accurate was the information retrieved?
- The generator: How well did the conversation flow?

This might sound simple at first glance, and frankly one of these evaluations kind of is. Testing a retriever is not a new concept in the world of AI and machine learning, and it actually has a name: **information retrieval**. Google has been doing it for decades to index the web, Amazon does it to find relevant products given a query, and librarians do it in person at your local library.

We started to tackle this problem in **Chapter 2** with our semantic search system by checking whether the top result retrieved was actually relevant. That was actually an example of the retriever's **precision**—the fraction of the documents that are retrieved in response to the query that are relevant, as seen in **Figure 4.9**.

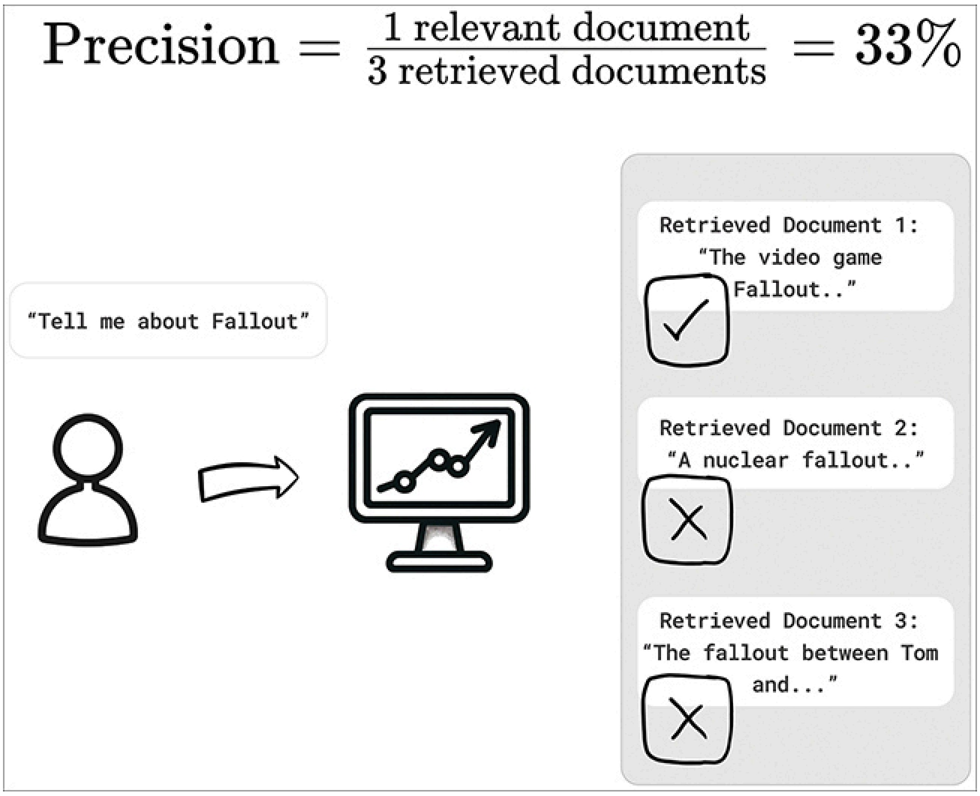

Figure 4.9 A RAG system's precision is a metric of "trust" revealing to us, on average, what % of the documents retrieved we can trust.

Precision has a built-in limitation, however: It assumes we could have multiple relevant documents per query. If we know that there's only one correct document per query, then precision will be relatively useless, because at best we will get only one correct document out of however many we retrieve. We will see more examples of evaluating retrieval systems in a later chapter when we design an end-to-end recommendation engine with fine-tuned LLMs.

On the generator side, we will tackle this task in more detail in **Chapter 12** on evaluating LLMs. For now, we note that it often boils down to evaluating the LLM's output either using a rubric, as visualized in **Figure 4.10**, or compared to a ground truth set, which will come back into play later in this book.

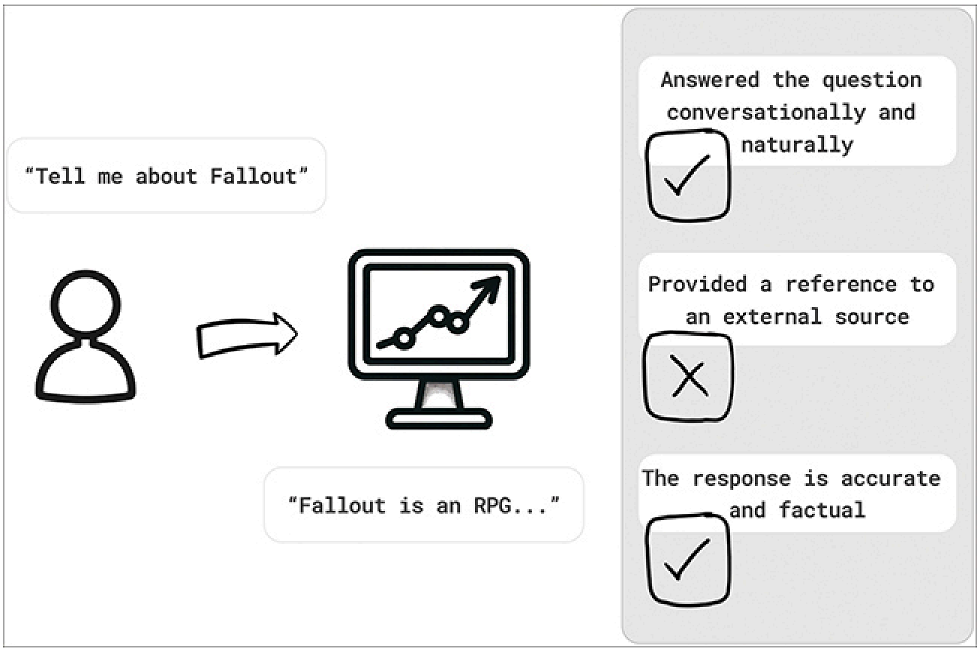

Figure 4.10 We can use a rubric to grade an LLM's generative response to give granular feedback that could be used in future fine-tuning loops.

It's easy to see why RAG systems can be quite powerful. They are a relatively easy way to ground an AI with facts from a database, and they rely more on an AI's reasoning and remixing power than its ability to recall encoded information from its parameters. Our RAG system had the ability to reach out via a tool to get some information and then use this information inline with a user conversation. When we used the combination of a one-shot example of a sample conversation and some chain of thought to force the AI to explain itself before actually answering, things looked pretty good.

What if information grabbed from a predefined database wasn't the only thing our AI had access to? What if we could give our AI a toolbox of tools to access, and let it decide which tool to use and how to use it? What if I stopped asking rhetorical questions and just went on to the next section?

Case Study 2: Automated AI Agents

Moving in the direction of popular AI frameworks and applications, the natural extension of a RAG system, with its ability to grab information and use it inline, is the idea of an "AI agent." I put that term in quotes because frankly it isn't really a technically defined term and different people implement these systems differently. Broadly speaking, an **AI agent** refers to an AI system with a generator (as in RAG) with access to multiple "tools" to accomplish tasks on behalf of the user. These tools range from looking information up—that would just be our RAG system—to writing and executing code, generating images, and checking my stock portfolio balance (all examples we will see in this chapter). Figure 4.11 shows the extremely high-level picture.

Figure 4.11 AI agents take in inputs from a user and utilize a tool from a toolbox to accomplish the task.

Popular frameworks like langchain have implementations of agents. I won't use any of them here, because my goal is to understand what's actually happening behind the scenes. What's happening behind the scenes is actually just some clever chain-of-thought prompting and few-shot learning.

Thought → Action → Observation → Response

There is no singular way to craft how an agent should behave. One popular method involves breaking down each query into four steps:

1. **Thought:** Force the generative components (GPT-3.5, in our example) to reason through what action to take based on the input.

2. **Action:** Have the AI decide both the action to take and any inputs to the action (e.g., the search query on Google).
3. **Observation:** Pass the response from the tool to the prompt so the generator can use it in context.
4. **Response:** Have the AI craft a response inline to the user using the context from the first three steps.

When the response is generated, the final output to the user will be natural, conversational, and usable. **Figure 4.12** zooms in on our agent architecture.

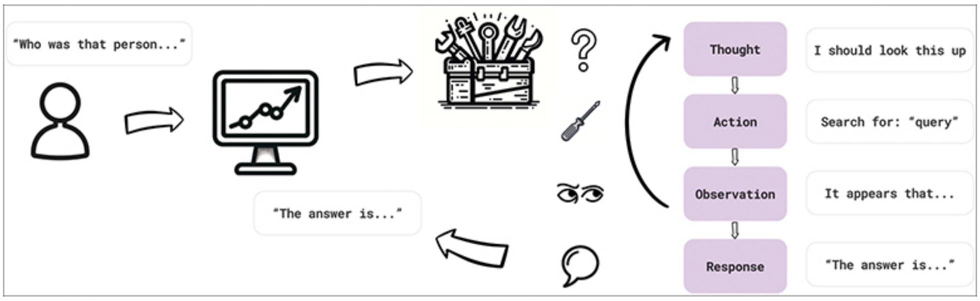

Figure 4.12 AI agents not only have to respond to the user, but also have to reason through many steps beforehand.

To actually achieve this thought pattern, we will write a prompt using both few-shot learning (one-shot, in this case) and chain-of-thought prompting (forcing the AI to walk through each step before responding). **Listing 4.2** shows the prompt we will use.

Listing 4.2 **Agent Prompt**

Click here to view code image

```
FINAL_ANSWER_TOKEN = "Assistant Response:"
OBSERVATION_TOKEN = "Observation:"
THOUGHT_TOKEN = "Thought:"
PROMPT_TEMPLATE = """Today is {today} and you can use tools to get new informa
Response the user's input as best as you can using the following tools:

{tool_description}
```

```
Use the following format:

User Input: the input question you must answer
Thought: comment on what you want to do next.
Action: the action to take, exactly one element of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
Thought: Now comment on what you want to do next.
Action: the next action to take, exactly one element of [{tool_names}]
Action Input: the input to the next action
Observation: the result of the next action
... (this Thought/Action/Action Input/Observation repeats until you are sure
answer)
Assistant Thought: I have enough information to respond to the user's input.
Assistant Response: your final answer to the original input question

Begin:

{previous_responses}
"""
```

This is essentially a more advanced version of our RAG prompt with more steps along the way to parse. Once our agent knows how to break down a task and pick a tool, we just need to give the AI some tools! In our code repository, I have about a half-dozen tools to use, including these:

- A **Python interpreter** to write and execute code via REPL (read, evaluate, print, and loop)
- API stock trading access via **Alpaca**
- Google searching via **SerpAPI**
- Image generation using **Stable Diffusion**

Listing 4.3 shows the basic tool interface class and the Python tool. For a complete list of tools, check out our repository.

Listing 4.3 **Python REPL Tool**

Click here to view code image

```
class ToolInterface(BaseModel):
    name: str
    description: str

    def run(self, input_text: str) -> str:
        # Must implement in subclass
        raise NotImplementedError("run() method not implemented")

class PythonREPLTool(ToolInterface):
```

```
    """A tool for running python code in a REPL."""

    globals: Optional[Dict] = Field(default_factory=dict, alias="_globals")
    locals: Optional[Dict] = Field(default_factory=dict, alias="_locals")

    name: str = "Python REPL"
    description: str = (
        "A Python shell. Use this to execute Python code. "
        "Input should be valid Python code. "
        "If you want to see the output of a value, you should print it out "
        "with 'print(...)'. Include examples of using the code and print "
        "the output."
    )

    def run(self, command: str) -> str:
        """Run command with own globals/locals and returns anything printed."""
        old_stdout = sys.stdout
        sys.stdout = mystdout = StringIO()
        try:
            exec(command, self.globals, self.locals)
            sys.stdout = old_stdout
            output = mystdout.getvalue()
        except Exception as e:
            sys.stdout = old_stdout
            output = str(e)
        return output

    def use(self, input_text: str) -> str:
        input_text = input_text.strip().replace("'''python" , "")
        input_text = input_text.strip().strip("'''")
        return self.run(input_text)
```

Once again, please check out the repository for the full commented code for these case studies. We can't fit all of it in this book, and most people don't like reading code on paper anyway. I get it. **Figure 4.13** visualizes this toolbox full of actual usable tools.

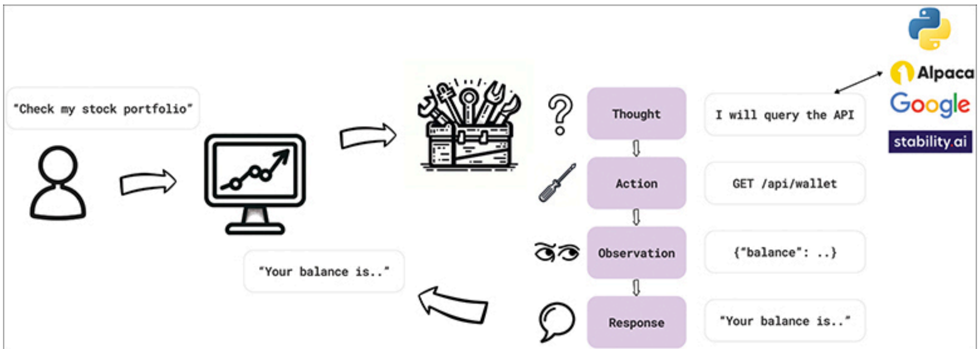

Figure 4.13 Our agent chooses which tool to use at every turn before responding to the user.

To showcase how our agent behaves with a sample conversation, **Figure 4.14** shows a real conversation I had with this agent, showing multiple tools being used.

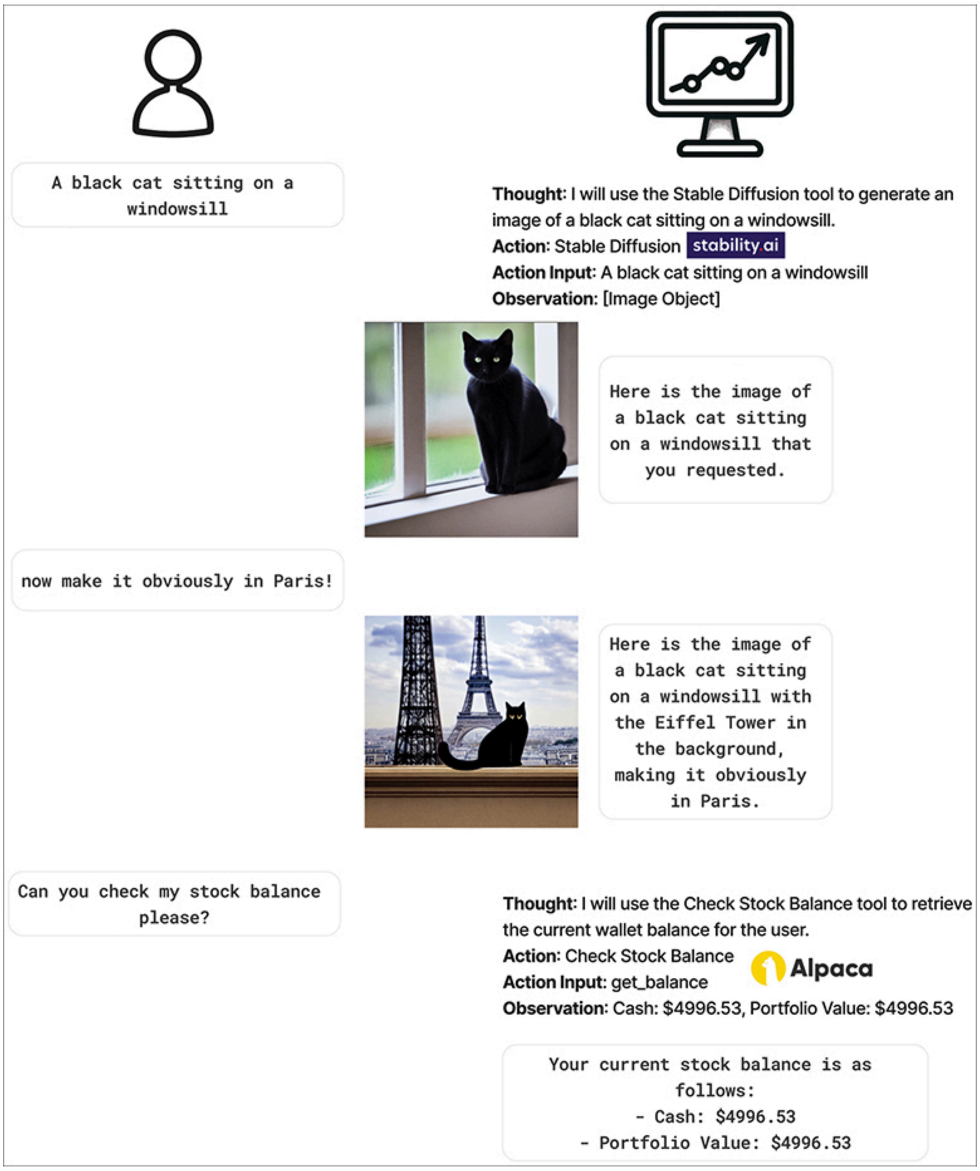

Figure 4.14 Holding a conversation with our agent shows off its ability to call upon different tools iteratively while still maintaining a conversational tone,

Agents really shine when they can chain together multiple thoughts in a row to solve a single question. Figure 4.15 shows an example of a single question triggering multiple thought → action → observation chains before a response is given.

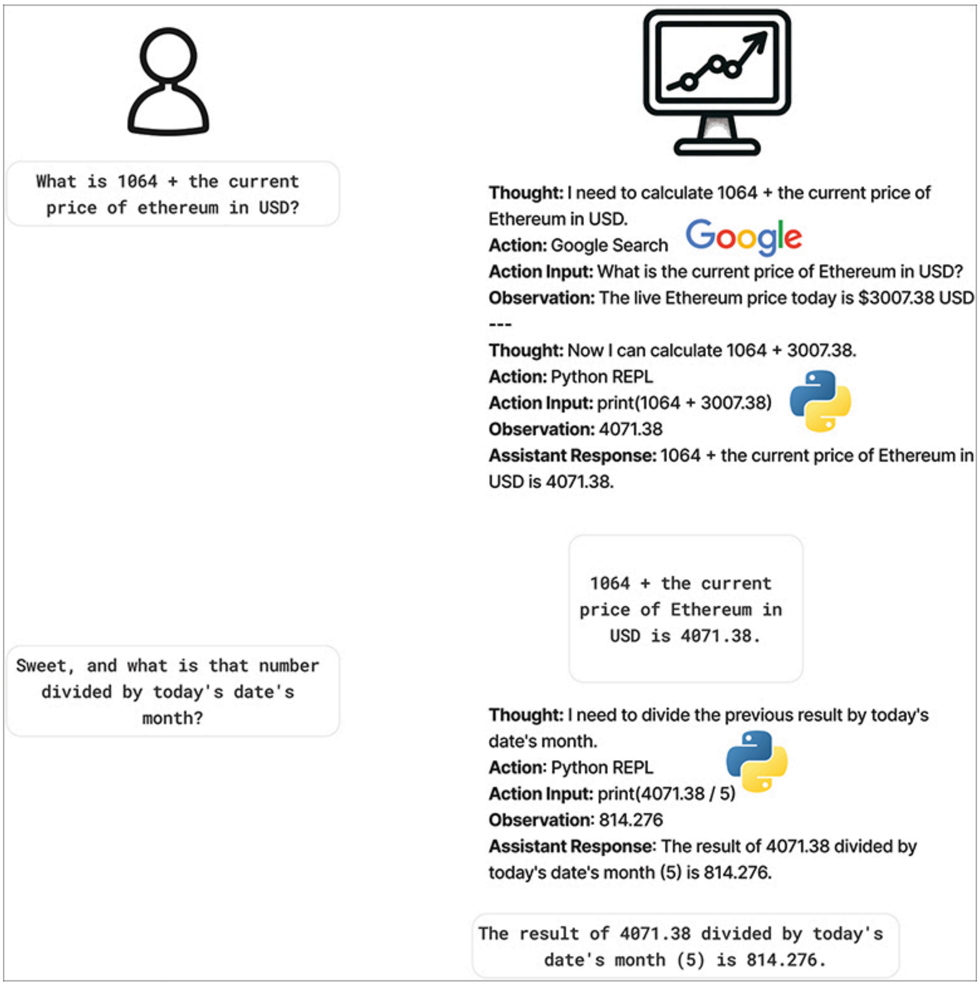

Figure 4.15 A decent AI agent can chain together multiple thoughts and tools in a single pass. In this case, it needed to both look up the current price of the cryptocurrency Ethereum (as of May 2024) and write Python code.

It's easy to look at these examples of conversations I've provided and say that it seems to be working well. In reality, gut checks like this are hardly ever enough to sign off on an agent for production.

Evaluating an AI Agent

Similar to evaluating our RAG system, evaluating our agent boils down to evaluating its ability to pick the right tool and create a decent response. Because our prompt involves more chain of thought, we could even begin to diagnose each individual thought process, as shown in Figure 4.16.

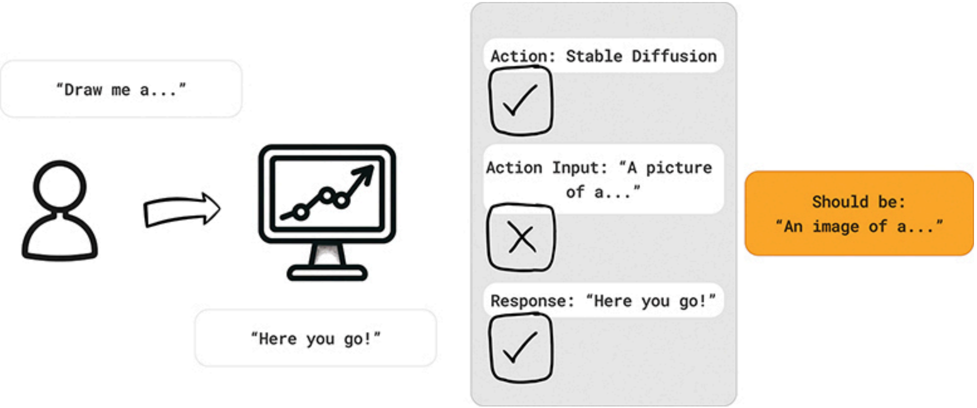

Figure 4.16 Evaluation of an AI agent can be as granular as dissecting and correcting each chain of thought in the series of steps.

Evaluating the performance of an AI system is crucial, but we are only scratching the surface here. Chapter 7 will deal with evaluations in much greater detail. It's important to start thinking about evaluation as soon as possible, and that often starts with understanding the individual components of your AI ecosystem and how each one can and should be tested.

Conclusion

As we wrap up the first part of this book, I want to do a quick debrief on what we have covered so far. In the next part of this book, we will begin to transition from the basics of using LLMs to more challenging applications, considerations, and nuances of deploying these models as prototypes, MVPs, and at scale.

The exploration of RAG systems and AI agents underscores a pivotal theme: the importance of context, adaptability, and a deep understanding of the tools at our disposal. Whether it's leveraging a database for grounding responses or orchestrating a symphony of digital tools to address user queries, the success of these applications hinges on a nuanced balance between the generative capabilities of LLMs and the specificity and reliability of external data sources and tools.

As we stand at this juncture, looking ahead to the next frontier of AI application, it's crucial to recognize that the journey is ongoing. The landscape of AI is perpetually evolving, with new challenges and opportunities emerging at the crossroads of technology and human needs. The insights garnered from the development and evaluation of RAG systems and AI agents are not endpoints, but rather stepping stones toward more sophisticated, empathetic, and effective AI applications.

In the chapters to come, we will delve deeper into the ethical considerations, the technical hurdles, and the uncharted territories of AI application. The goal is not just to build AI systems that work, but to create experiences that enhance human capabilities, foster understanding, and, ultimately, enrich lives.

The AI ecosystem is vast and varied, filled with potential and pitfalls. Yet, with a thoughtful approach and a clear vision, the pieces can come together to form solutions that are not just technically proficient but also meaningful and impactful. This is the essence of AI application—a journey of discovery, creativity, and continuous improvement.