CHAPTER 4

Designing LLM Applications

The previous two chapters laid the foundations for the remainder of the book. Chapter 2 showed in detail how LLMs function, and we demonstrated that at the end of the day, they are effectively document completion models that predict content one token at a time. Chapter 3 explained how the chat API is built upon the LLMs of Chapter 2. With some syntactic sugar at the API level and a healthy dose of fine-tuning, the document completion model is used to complete conversations between the user and an imagined assistant. When you get down to it, the chat model is really still a document completion model—it's just that the documents it completes are all conversation transcripts.

From this point forward in the book, you'll learn everything you need to know about how to build LLM applications to solve problems on behalf of your company and your users. This chapter serves as a gateway to that content. In this chapter, we'll dive into the LLM *application*, which you'll see is actually a transformation layer between the user's problem domain and the model's text domain. Furthermore, the LLM application is a transformation layer with a purpose—solving problems!

The Anatomy of the Loop

In Figure 4-1, the LLM application is represented as a *loop*, meaning an interaction back and forth between the user and the model. The domains of the model and the user are often quite different. The user may be doing any number of things, such as writing an email and looking for just the right wording to communicate their point. Or they may be doing something complicated, such as organizing group travel, booking travel tickets, and procuring lodging. Perhaps the user isn't directly in contact with the LLM application; for instance, they could have set up a recurring analysis that the LLM application performs periodically as new data becomes available. The point is that the user can be doing a great variety of things.

The model, on the other hand, does only one thing—it completes documents. But this capability affords you a great deal of flexibility when building the LLM application. The ability to complete documents gives the model the ability to write emails, code, stories, documentation, and (in principle) anything else that a human might write. As we showed in the previous chapter, a chat app is an LLM application that completes *transcript* documents, and tool execution is simply going one step farther and completing a specialized transcript document that includes a function calling syntax. With their ability to complete text, engage in chat, and execute tools, LLMs can be applied to an almost unlimited number of use cases.

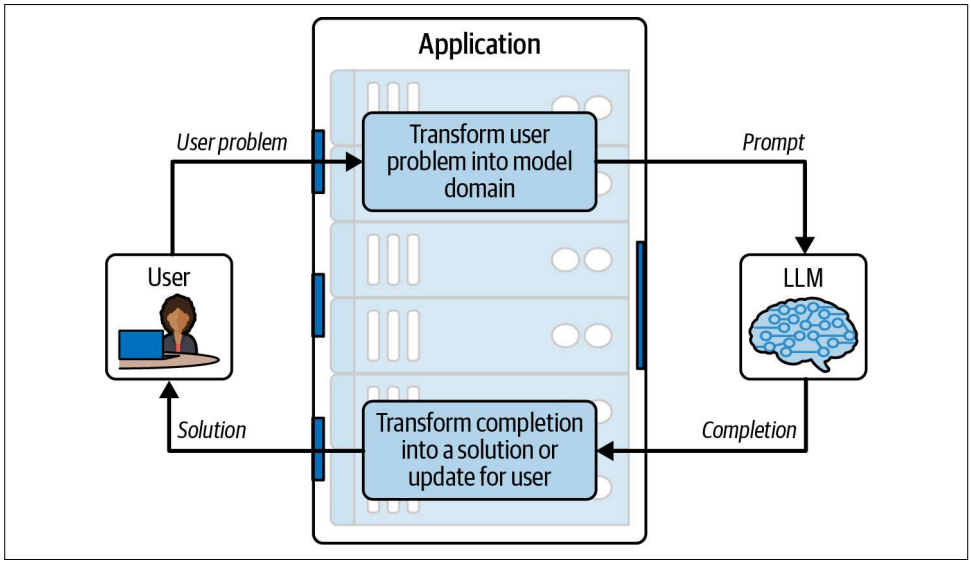

Figure 4-1. LLM-based applications implement the loop, which conveys information from the user domain to the LLM's text domain and then back

The loop implements the transformation between the user domain and the model domain. It takes the user's problem and converts it into the document or transcript that the model must complete. Once the model has responded, the loop transforms the model output back into the user domain in the form of a solution to the user's problem (or at least a step in the right direction).

The LLM application may involve just one iteration of the loop. For instance, if the user is writing an email and wants to convert a bulleted list of points into prose, then you need only one iteration through this loop—once the model returns the prose, the job of the application is complete. The user can run the application again if they want, but in each case, the loop retains no state from the previous run.

Alternatively, the LLM application may run the loop several times in a row, as is the case for a chat assistant. Or, the LLM application may run iteratively, refer to a vast

amount of state, and modify the loop as the problem changes shape. A travel planning app is a good example of this. Initially, the application would help brainstorm travel ideas; then, it would move on to making the actual travel arrangements; and finally, it would set up reminders and travel tips.

In the following sections, we'll take you for one trip around the loop of Figure 4-1. We will discuss the user's problem domain, convert that problem to the model domain, collect the completion, and convert it back into a solution for the user.

The User's Problem

The loop starts with the user and the problem they are trying to solve. Table 4-1 illustrates how the user's problem domain can vary among several dimensions and can range from simple to complex. These dimensions include the following:

- The medium in which the problem is conveyed (with text being the most natural for LLMs)
- The level of abstraction (with higher abstraction requiring more complex reasoning)
- The context information required (with most domains requiring retrieval of additional information besides what is supplied by the user)
- How stateful the problem is (with more complex problem domains requiring memory of past interactions and user preferences)

As you can see in Table 4-1, user problem domains have various levels of complexity in several dimensions. For example, a proofreading application would be low complexity in all dimensions, while a travel planning assistant would be quite complex. When you're building an LLM application, you'll deal with all these forms of complexity in different ways. We'll give you a glimpse into these approaches in this chapter and then greater detail on them throughout the rest of this book.

Table 4-1. Three problem domains (in the columns) in four dimensions of complexity (in the rows)

	Increasing complexity →		
	Proofreading	**IT support assistance**	**Travel planning**
Medium of the problem	Text	Voice over the phone	Complex interactions on the website, text input from the user, and interactions with APIs.
Level of abstraction	The problem is concrete, well-defined, and small.	A large abstract problem space and a large solution space, but constrained by available documentation	The problem involves understanding the user's subjective tastes and objective constraints in order to coordinate a complex solution.

	Increasing complexity ➜		
	Proofreading	**IT support assistance**	**Travel planning**
Context required	Nothing more than the text submitted by the user.	Searchable access to technical documentation and example support transcripts	Access to calendars, airlines APIs, recent news articles, government travel recommendations, Wikipedia, etc.
Statefulness	No statefulness—every call to the API contains a distinct problem statement.	Must track the conversation history and solutions attempted	Must track interaction across weeks of planning, different mediums of interaction, and aborted branches of planning.

Converting the User's Problem to the Model Domain

The next stop on the loop from Figure 4-1 is inside the application, where the user's problem is converted into the domain of the model. The crux of prompt engineering lies in this step. The goal is to create a prompt so that its completion contains information that can be used to address the user's problem. Crafting just the right prompt is quite a tall order, and the application must satisfy the following criteria simultaneously:

1. The prompt must closely resemble content from the training set.
2. The prompt must include all the information relevant to addressing the user's problem.
3. The prompt must lead the model to generate a completion that addresses the problem.
4. The completion must have a reasonable end point so that generation comes to a natural stop.

Let's dig into each of these criteria. First and foremost, the prompt must closely resemble documents from the training set. We call this the *Little Red Riding Hood principle*. You remember that story, right? A naive girl dressed in fashionable red attire walks along a forest path to visit her ailing grandmother. Despite her mother's stern warnings, the girl strays from the path and has an encounter with a wolf (big and bad), and then the story really goes south—much gore...*much gore*. It's really crazy that we tell this story to children.

But for our purposes, the point is simple: don't stray far from the path upon which the model was trained. The more realistic and familiar you make the prompt document and the more similar it is to documents from the training set, the more likely it is that the completion will be predictable and stable. The Little Red Riding Hood principle is one that we will revisit several times in this book. For now, suffice to say that you should always mimic common patterns found in training data.

Most of the best LLMs are tight-lipped about their training data, and for good reason. If you know exactly how their training documents are formatted, then you have a leg up on manipulating the prompt and, say, finding a new jailbreaking strategy. However, if you want to see what kinds of documents the models are familiar with, then the easiest thing to do is—*just ask*. As an example, try this request: **`"What types of formal documents are useful for specifying financial information about a company?"`** You should see a large selection of documents to pattern your request after. Next, ask the model to generate an example document and see if it's what you need.

Fortunately, there are endless types of documents and motifs to draw from. For completion models, see if you can make the prompt resemble computer programs, news articles, tweets, markdown documents, communication transcripts, etc. For chat models, the overall document is decided for you—for OpenAI, this is a ChatML document that starts with an instructive system message followed by back-and-forth exchanges between the user and the assistant character. But you can still use the Little Red Riding Hood principle by including common motifs within the user messages. For instance, make use of markdown syntax to help the model understand the structure of the content. Use a hash sign (`#`) to delimit sections, backticks (```` ``` ````) to delimit code, an asterisk (`*`) to indicate items in a list, etc.

Now, let's look at the second criterion: the prompt must include all the information relevant to addressing the user's problem. As you convert the user's problem into the model's domain, you must collect all of the information relevant to solving the user's problem and incorporate it into the prompt. Sometimes, the user directly supplies you with all of the information that you need—in the proofreading example, the user's raw text is sufficient. But at the other extreme, the travel planning application requires that you pull in user preferences, information from user calendars, airline ticket availability, recent news about the destination, government travel recommendations, etc.

Finding all the *possible* content is one challenge, and finding the *best* content is the next challenge. If you saturate the prompt with too much loosely relevant content, then the language model will get distracted and generate irrelevant completions. Finally, the content must be arranged in a well-formatted, logical document so that it makes sense—lest you stray off the path on the way to Grandmother's house.

The third criterion to consider is that the prompt must condition the model to generate a completion that is actually helpful. If the LLM continues after the prompt by merely jabbering on about the user's problem, then you're not helping them at all. You must therefore carefully consider how to set up the prompt so that it points to a solution. When working with completion models, this can be surprisingly tricky. You

will need to let the model know that it's time to create the solution (see the homework example that follows). For chat models, this is much easier because the model has been fine-tuned to automatically produce a helpful message from the assistant that addresses the user problem. Thus, you don't need any trickery to pull an answer out of the model.

Finally, you must ensure that the model actually stops! Here again, the situation is different for completion versus chat models. With chat, everything is easy—the model is fine-tuned to come to a stop after the helpful assistant message (though you might need to instruct the assistant to limit how chatty it is). With completion models, you have to be more careful. One option is to create an expectation in the instructional text that the solution should *not* go on forever; it should reach a solution and stop. An alternative is to create the expectation that some specific thing will follow and that it will begin with very specific and easily identifiable opening text. If such a pattern exists, then we can use the `stop` parameter to halt generation at the moment the opening text is produced. Both of these patterns are seen in the example covered next.

So a Funny Thing Happened...

At GitHub, in the early days of the chat models, we made a funny mistake. The models are fine-tuned to end assistant messages with the special `<|im_end|>` token and then halt generation. This is great—it means you don't have to do anything special to ensure that the model will stop. But we had configured this particular model incorrectly, causing it to suppress the `<|im_end|>` token. Amusingly, we ended up with a model that literally didn't know how to shut up. It would begin with a very intelligible answer from the assistant, and then, it would end with a salutation, "Hope you have a nice day!" But then, since it *literally* couldn't stop, it had to think of something to say next. So it continued, "Hope you have a wonderful day!" and "Hope you have a festive day!" and so on, and so on, until it had found all the synonyms available for *wonderful* and was finally forced to stop at the token limit.

Example: Converting the user's problem into a homework problem

Let's dig into an example to demonstrate the preceding concepts. Table 4-2 shows an example prompt for an application that makes travel recommendations based on a user's requested location. The plain text is part of the boilerplate used to structure the prompt and condition it to provide a solution, and the italicized text is the information specific to the user's current request. This example uses a completion API because it makes it easier to see each of the preceding criteria in action. (Note that building an actual travel app would be very complicated indeed! We chose this very simplified example because it demonstrates the ideas discussed previously. We talk about more realistic applications in Chapters 8 and 9.)

Table 4-2. An example prompt for a travel recommendation application

Prompt	`# Leisure, Travel, and Tourism Studies 101 - Homework Assignment`
	
	`Provide answers for the following three problems. Each answer should be concise, no more than a sentence or two.`
	
	`## Problem 1`
	`What are the top three golf destinations to recommend to customers? Provide the answer as a short sentence.`
	
	`## Solution 1`
	`St. Andrews, Scotland; Pebble Beach, California; and Augusta, Georgia, USA (Augusta National Golf Club) are great destinations for golfing.`
	
	`## Problem 2`
	`Let's say a customer approaches you to help them with` *`travel plans for Pyongyang, North Korea.`*
	
	`You check the State Department recommendations, and they advise` *`"Do not travel to North Korea due to the continuing serious risk of arrest and long-term detention of US nationals. Exercise increased caution in travel to North Korea due to the critical threat of wrongful detention."`*
	
	`You check the recent news and see these headlines:`
	`- "North Korea fires ballistic missile, Japan says"`
	`- "Five-day COVID-19 lockdown imposed in Pyongyang"`
	`- "Yoon renews efforts to address dire North Korean human rights"`
	
	`Please provide the customer with a short recommendation for travel to their desired destination. What would you tell the customer?`
	
	`## Solution 2`

Completion	`Perhaps North Korea isn't a great destination right now.` `But I bet we could find some nice place to visit in South Korea.`

First, notice how the prompt obeys the Little Red Riding Hood principle—this is a homework problem, a type of document that you are likely to find regularly in training data. Moreover, the document is formatted in Markdown, a common markup language. This will encourage the model to format the document in a predictable way, with section headings and syntax indicating bold or italicized words. At the most basic level, the document uses proper grammar. This is important, as sloppy grammar will encourage the model to generate text in a similar, sloppy style. Clearly, we are solidly on the path to Grandmother's house.

Next, take a look at how the prompt incorporates the context that the LLM will need to understand the problem; this context appears in italics in Table 4-2. First is the actual user problem. Probably, the user has just selected North Korea from a

drop-down menu on the travel website; they may have even selected it by mistake. Nevertheless, it is added to the prompt as the first bold text snippet. The subsequent scraps of bold text are pulled from other relevant resources: State Department travel recommendations and recent news article headings. For our example, this is enough information to make a travel recommendation.

There are several ways in which this prompt leads the model toward a definite solution, rather than toward further elaboration of the problem. In the first line, we condition the model toward the type of response we hope to see—something within the domain of leisure, travel, and tourism. Next, we include an example problem. This has nothing to do with the user's current request, but it establishes a pattern for the model: the problem will begin with `## Problem N` and will be followed by a solution starting with `## Solution N`.

Problem 1 also encourages the use of a certain voice for the subsequent answers—concise and polite. The fact that solution 1 is a short sentence further encourages the continuation of this pattern in the completion. With this pattern in place, problem 2 is the actual user problem. We set up the problem, insert the context, and make the ask: `What would you tell the customer?` With the text `## Solution 2`, we then indicate that the problem statement is over and it's time for the answer. If we had omitted this, then the model would likely have continued elaborating upon the problem by confabulating more information about North Korea.

The last task is to insist upon a firm stop. Since every new section of markdown begins with ##, we have a pattern that we can capitalize upon. If the model begins to confabulate a third problem, then we can cut off the model completion by specifying stop text, which tells the model to halt generation as soon as this text is produced. In this case, a reasonable choice for stop text is `\n#`, which indicates that the model has completed the current solution and is beginning a new section, possibly the start of a confabulated problem 3.

Chat models versus completion models

In the preceding example, we've relied on a completion model to demonstrate the criteria for converting between the user domain and the model domain. With the introduction of chat models, much of this is simplified. The chat APIs ensure that the input into the models will closely resemble the fine-tuning data because the messages will be internally formed into a transcript document (criterion 1 from the beginning of this section). The model is highly conditioned to provide a response that addresses the user's problem (criterion 3), and the model will always stop at a reasonable point—at the end of the assistant's message (criterion 4).

But this doesn't mean that you, as the prompt engineer, are off the hook! You're fully responsible for including all the relevant information for addressing the user's problem (criterion 2). You must craft the text within the chat so that it resembles

characteristics of documents in training (criterion 1). Most importantly, you must shape the transcript, system message, and function definitions so that the model can successfully address the problem and come to a stopping point (criteria 3 and 4).

Now, You Try!

Using a completion model such as gpt-3.5-turbo-instruct, start with the preceding prompt and see what happens as you modify pieces of the prompt in these ways:

1. What if you leave off `## Solution 2` or even the question that precedes it? Does the model continue to elaborate on the problem statement? Even if the model completes the problem statement, why is it still important to keep the question and the solution heading?
2. Problem 1 serves as an example. If you change the solution 1 text, does it modify the text generated for solution 2? Try increasing or decreasing solution 1's length significantly. Try making it talk like a pirate. Try making it rude. How do those modifications affect solution 2?
3. Try keeping the same country but replacing the negative context with increasingly positive remarks. Does the model still recommend against travel to North Korea? Why might this be?
4. If you omit the stop word, then does the model confabulate a third problem? If not, then what if you add one more new line character? Can you introduce one character to make it confabulate a fourth problem?
5. Are there any reasons that using a homework problem might be problematic? Try a different format, such as a transcript of a travel agency help hotline.

Using the LLM to Complete the Prompt

Referring back to Figure 4-1, in the next stage of the LLM-application loop, you submit the prompt to the model and retrieve the completion. If you've played with only one particular model, such as ChatGPT, you might be under the impression that there are no decisions to make here—just send the model a prompt and wait for the completion, just as we showed in the example. However, all models are *not* alike!

You'll have to decide how big your model should be. Typically, the larger the model is, the higher quality its completions will be. But there are some very important trade-offs, such as cost. At the time of writing this book, running GPT-4 can be *20 times* more expensive than running gpt-3.5-turbo. Is the quality improvement worth the order-of-magnitude increase in price? Sometimes, it is!

Also of importance is the latency. Bigger models require more computation, and more computation might require more time than your users can spare. In the early days of GitHub Copilot, we decided to use an OpenAI model called Codex, which is

small, *sufficiently* smart, and lightning fast. If we had used GPT-4, then users would have rarely been inclined to wait for the completion, no matter how good it was.

Finally, you should consider whether or not you can gain better performance through fine-tuning. At GitHub, we're experimenting with fine-tuning Codex models to provide higher quality results for less common languages. In general, fine-tuning can be useful when you want the model to provide information that is unavailable in the public datasets that the model was originally trained on, or when you want the model to exhibit behavior that is different from the behavior of the original model. The process of fine-tuning is beyond the scope of this book, but we're confident that fine-tuning models will become simpler and more commonplace, so it's definitely a tool you should have in your belt.

Transforming Back to User Domain

Let's dig into the final phase of the loop from Figure 4-1. The LLM completion is a blob of text. If you're making a simple chat app of some sort, then maybe you're done—just send the text back to the client and present it directly to the user. But more often, you will need to transform the text or harvest information from it to make it useful to the end user.

With the original completion models, this often meant asking the model to present specific data with a very specific format and then to parse that information out and present it back to the user. For instance, you might have asked the model to read a document and then generate tabular information that would have been extracted and represented back to the user.

However, since the appearance of function-calling models, converting model output into information that's useful to the user has become quite a bit easier. For these models, the prompt engineer lays out the user's problem, gives the model a list of functions, and then asks the model to generate text. The generated text then represents a function call.

For instance, in a travel app, you might provide the model with functions that can look up airline flights and a description of a user's travel goals. The model might then generate a function call requesting tickets for a particular date with the user's requested origin and destination. An LLM application can use this to call the actual airline's API, retrieve available flights, and present them to the user—back in the user's domain.

You can go further by giving the model functions that actually create a change in the real world. For instance, you can provide the model with functions that actually purchase tickets. When the model generates a function call to purchase tickets, the application can double-check with the user that this is OK and then complete the transaction. Thus, you have translated from the model domain—text representing a

function call—to the user domain in the form of an actual purchase on the user's behalf. We will go into more detail about this in Chapters 8 and 9.

Finally, when transforming back to the user domain, you may change the medium of communication entirely. The model generates in text, but if the user is speaking to an automated tech support system over their phone, then the model completions will need to be converted into speech. If the user is using an application with a complicated UI, then the model completions might represent events that modify elements of the UI.

And even if the user's domain is text, it might still be necessary to modify the presentation of the model completions. For instance, Copilot code completion is represented as a grayed-out code snippet in the IDE, which the user can accept by pressing Tab. But when you use Copilot chat to ask for a code change, the results are presented as a red/green text diff.

Zooming In to the Feedforward Pass

Let's spend some more time examining the LLM-application loop from Figure 4-1—specifically, the *feedforward pass*, which is the part of the loop where you convert the user problem into the domain of the model. Almost all of the remaining chapters in this book will go into great detail about just *how* we achieve high-quality completions. But before we get into the nitty-gritty, let's lay down some foundational ideas that we'll build on in coming chapters.

Building the Basic Feedforward Pass

The feedforward pass is composed of several basic steps that allow you to translate the user's problem into the text domain (see Figure 4-2). The middle chapters of this book will cover these steps in detail.

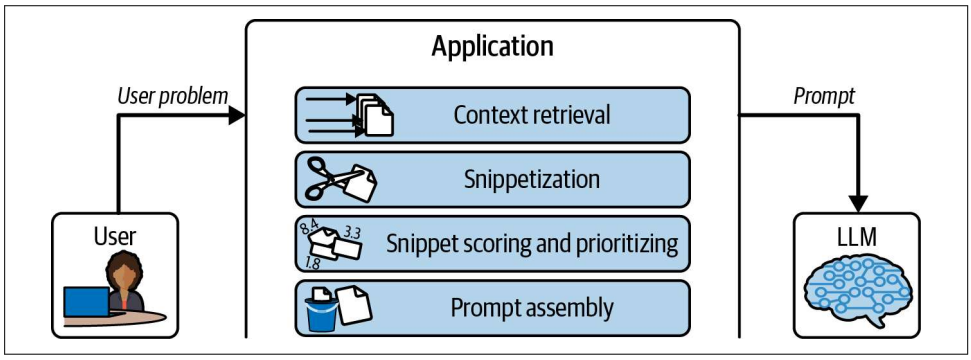

Figure 4-2. Typical basic steps for translating the user's problem into the domain of the LLM

Context retrieval

The first thing you do to build the feedforward pass is create or retrieve the raw text that serves as the context information for the prompt. One way to think through this problem is to consider context in terms of how *direct* or *indirect* it is.

The *most direct* context comes straight from the user as they describe their problem. If you're building a tech support assistant, this is the text that the user types directly into the help box; with GitHub Copilot, this is the code block that the user is editing right now.

Indirect context comes from relevant sources nearby. If you're building a tech support app, for example, you might search documentation for excerpts that address the user's problem. For Copilot, the indirect context comes largely from other open tabs in the developer's IDE because these files often include snippets relevant to the user's current problem. The least direct context corresponds to the boilerplate text that is used to shape the response of the model. For a tech support app, this could be the message at the top of the prompt that says, "This is an IT support request. We do whatever it takes to help users solve their problems."

Boilerplate text at the top of the prompt is used to introduce the general problem. Later in the prompt, it acts as a glue to connect the bits of direct context in such a way that it makes sense to the model. For instance, the nonbolded text in Table 4-2 is boilerplate. The boilerplate at the top of the table introduces the travel problem, and the boilerplate farther down allows us to incorporate information directly from the user regarding travel plans as well as relevant information pulled from news and government sources.

Snippetizing context

Once the relevant context has been retrieved, it must be snippetized and prioritized. *Snippetizing* means breaking down the context into the chunks most relevant for the prompt. For instance, if your IT support application issues a documentation search and returns with pages of results, then you must extract only the most relevant passages; otherwise, we may exceed the prompt's token budget.

Sometimes, snippetizing means creating text snippets by converting context information from a different format. For instance, if the tech support application is a phone assistant, then you need to transcribe the user's request from voice to text. If your context retrieval calls out to a JSON API, then it might be important to format the response as natural language so that the model will not incorporate JSON fragments into its response.

Scoring and prioritizing snippets

The token window of the original GPT-3.5 models was a measly 4,096 tokens, so running out of space was once a pressing concern in any LLM application. Now, with token windows upward of 100,000 tokens, it's less likely that you'll run out of space in your prompt. However, it's still important to keep your prompts as trim as possible because long blobs of irrelevant text will confuse the model and lead to worse completions.

To pick the best content, once you've gathered a set of snippets, you should assign each snippet either a priority or a score corresponding to how important that snippet will be for the prompt. We have very specific definitions of scores and priorities. *Priorities* can be thought of as integers that establish tiers of snippets based upon how important they are and how they function in the prompt. When assembling the prompt, you'll make sure that all snippets from a higher tier are utilized before dipping into the snippets from the next tier. *Scores*, on the other hand, can be thought of as floating-point values that emphasize the shades of difference between snippets. Some snippets within the same priority tier are more relevant than others and should be used first.

Prompt assembly

In the last step, all of this snippet fodder gets assembled into the final prompt. You have many goals during this step: you must clearly convey the user's problem and pack the prompt as full of the best supporting context as possible—and you must *make sure* not to exceed the token budget, because all you'll get back from the model in that case is an error message.

It's at this point where accounting comes heavily into play. You must make sure that all your boilerplate instructions fit in the prompt context, make sure that the user's request fits, and then collect as much supporting context as possible. Sometimes, during this step, you might want to make a last-minute effort to shorten the context. For instance, if you know that a full code file is relevant to the user's answer but doesn't fit, you have an option during this step to elide (remove) less relevant lines of code until the document fits. If you have a long document, you can also employ summarization.

In addition to making sure all the pieces fit, you must ensure they are assembled into their proper order. Then, the final prompt document should read like a document you might find in the training data (leading Little Red Riding Hood on the path directly to Grandma's house).

Exploring the Complexity of the Loop

The previous section focused on the simplest type of LLM application—one that does all of its work in a single request to the model and then returns the completion to

the user. Such a simple application is important to understand because it serves as the starting point. It presents basic principles upon which applications of increasing complexity are built. As applications get more complex, there are several dimensions along which this complexity comes into play:

- More application state
- More external content
- More complex reasoning
- More complex interaction with the world outside of the model

Persisting application state

The feedforward application from the previous section holds no persistent state. It simply takes the user's input, adds on some *hopefully* relevant context, passes it on to the model, and then passes the model's response back to the user. In this simple world, if the user makes another request, the application has no recollection of the previous exchange. Copilot code completion is an application that works exactly this way.

More complex LLM applications usually require state to be maintained between requests. For instance, even the most basic chat application must maintain a record of the conversation. During the middle of a chat session, when the user submits a new message to the application, the application looks up this conversation thread in a database and uses the previous exchanges as further context for the next prompt.

If a user's interactions are long running, then you may need to abridge the history to fit it into the prompt. The easiest way to accomplish this is by just truncating the conversation and cutting off the earlier exchanges. This won't always work, though! Sometimes, the content is too important to cut, so another approach is to summarize earlier parts of the conversation.

External context

LLMs—even the best ones—don't have *all* the answers. How could they? They've been trained only on publicly available data, and they have no clue about recent events and information that is hidden behind a corporate, government, or personal privacy wall. If you ask a model about information that it does not possess, then *ideally*, it will apologize and explain that it doesn't have access to that information. This doesn't lead to user satisfaction, but it's infinitely better than the alternative—the model confidently hallucinating an answer and telling the user something that is completely false.

For this reason, many LLM applications employ retrieval augmented generation (RAG). With RAG, you augment the prompt with context drawn from sources that

were unavailable to the model during training. This could be anything from your corporate documentation to your user's medical records to recent news events and recently published papers.

This information is indexed into a search engine of some sort. Lots of people have been using embedding models to convert documents (or document fragments) into vectors that can be stored in a vector store (like Pinecone). However, you shouldn't turn up your nose at good old-fashioned search indexes (such as Elasticsearch) because they tend to be relatively simple to manage and much easier to debug with when you don't seem to be finding the documents that you're looking for.

Actually retrieving the context usually follows a spectrum of possible approaches. The simplest is to directly use the user's request as the search query. However, if your user's request is a long run-on paragraph, then it might have extraneous content that causes spurious matches to come back from the index. In this case, you can ask the LLM what it thinks a good search will be and just use its response text to search the index. Finally, if your application is in some sort of long chat with a user, it might not at all be apparent when it's worth even searching for something; you can't retrieve documents for every comment they have because they might still be talking about documents related to their last comment. In this case, you can introduce a *search tool* to the assistant and let the assistant choose when to make a search and what search terms to use. (We'll introduce tool usage just a bit further on.)

Increasing reasoning depth

As we covered in Chapter 1, the really spectacular thing about the larger LLMs starting with GPT-2 was that they began to generalize much more broadly than their predecessors. The paper entitled "Language Models are Unsupervised Multitask Learners" makes just this point—GPT-2, trained on millions of web pages, was able to beat benchmarks in several categories that had until that point required very specialized model training.

For instance, to get GPT-2 to summarize text, you could append the string **`TL;DR`** to the end of the text, et voilà! And to get GPT-2 to translate text from English to French, you could just provide it with one example translation and then subsequently provide the English sentence to be translated. The model would pick up on the pattern and translate accordingly. It was as if the model were actually in some way *reasoning* about the text in the prompt. In subsequent years, we've found ways to elicit more sophisticated patterns of reasoning from the LLMs. One simple but effective approach is to insist that the model show its step-by-step thought process *before* providing the answer to the problem. This is called *chain-of-thought* prompting. The intuition behind this is that, unlike humans, LLMs have no internal monologue, so they can't really think *about* a problem before answering.

Instead, each token is mechanically generated as a function of every token that preceded it. Therefore, if you want to have the model "think" about a problem before answering, the thinking must be done "out loud" in the completion. Afterward, when subsequent tokens are calculated, the model will predict tokens that are as consistent as possible with the preceding tokens and therefore consistent with their "thought process." This often leads to much better-reasoned answers.

As LLM applications require more complicated work to be completed, the prompt engineer must find clever ways to break the problem down and elicit the right step-by-step thinking for each component to drive the model to a better solution.

Tool usage

By themselves, LLMs act in a closed world—they know nothing about the outside world and have no ability to effect change in the outside world. This constraint seriously limits the utility of LLM applications. In response to this weakness, most frontier LLMs are now able to interact with the world through *tools*.

Take a look at the *tool loop* in Figure 4-3. The idea is simple. In the prompt, you make the model aware of one or more tools that it has access to. The tools will look like functions including a name, several arguments, and descriptions for the name and arguments. During a conversation, the model can choose to execute these tools—basically by calling one of the functions with an appropriate set of arguments.

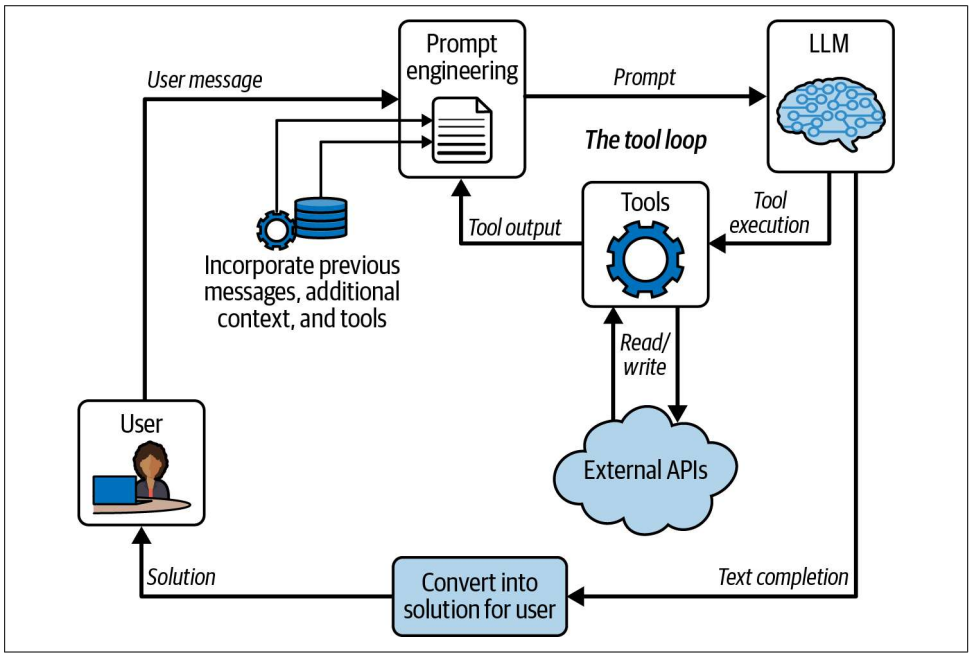

Figure 4-3. A more complicated application loop that includes an internal tool loop

Note that LLM-applications can become quite complex. Conversations are stateful, and the context must be preserved from one request to the next. Information from external APIs is used to augment the data, and the tool execution loop may iterate several times back and forth between the application and the model before information can be returned to the user.

Naturally, the model has no ability to actually execute code, so it is the responsibility of the LLM application to intercept this function call from the model and execute a real-world API and the appended information from the response to the prompt. Because of this, on the next turn, the model can use that information to reason about the problem at hand.

One of the earlier papers to consider tool usage was "ReAct: Synergizing Reasoning and Acting in Language Models" (2022). It introduced three tools: `search`, `lookup`, and `finish`, which, respectively, allowed the model to search through Wikipedia, look up relevant blocks of text within a Wikipedia page, and return the answer to the user. This shows how tool usage can overlap with RAG—namely, if you provide the model with search tools, it will be able to make its own determination of when it needs external information and how to find it.

Search, though, is a read-only behavior. Similarly, tools connected to external APIs that check the temperature, determine if you have any new emails, or retrieve recent LinkedIn posts are all read-only. Where things get really interesting is when we allow them to write changes out into the real world. Since tools give models access to any real-world API imaginable, you'll be able to create LLM-based assistants that can write code and create pull-requests, help you plan travel and reserve airfare and lodging, and so much more. Naturally, *with great power comes great responsibility*. Models are probabilistic and *often* make mistakes, so don't let the LLM application book a trip to Greece just because the user said they would love to visit someday!

Evaluating LLM Application Quality

Again, we say that LLMs are probabilistic and *often* make mistakes. Therefore, when designing and productionizing an LLM application, it is imperative that you constantly evaluate application quality. Before you ship a new LLM-based feature, take time to prototype the functionality and gather some quantitative metrics about how the model will react. And then, once a feature ships, your application should be recording telemetry so that you can keep an eye on both the model's and the users' behavior so that you can quickly ascertain any degradation in the quality of the application.

Offline Evaluation

Offline evaluation is all about trying new ideas for your LLM application *before* exposing your users to an untested new experience. If anything, offline evaluation is even more complex than online evaluation, which we described later in this section. Since, before shipping a feature to production, you don't have any customers to tell you "good" or "bad," you have to figure out some simulated proxy for this evaluation.

Sometimes, you get lucky. For example, with Copilot code completions, a good proxy for user satisfaction is whether or not the code is functional and complete. In the case of code, this is actually quite easy to measure—if you can delete fragments of working code and then generate a completion that still passes the tests, then the code works and your users will likely be happy with similar completions in production. This is exactly how we evaluated changes prior to shipping them—we grabbed a few hundred repos, made sure their tests ran, surgically deleted and generated fragments of code, and then saw whether or not the tests still ran.

Often, you won't be this lucky. How do you evaluate a scheduling assistant that is expected to create real-world interactions, and how do you evaluate a general chat application that engages users in open-ended dialogue? One emerging approach is to make an LLM act as a judge, much like a human judge, and review chat transcripts and determine which variant is best. The judgment can be an answer to a basic question like "Which version is better?" However, for a more nuanced score, you can give the judge a checklist of criteria to review for each variant.

However you choose to evaluate your LLM application, always try to engage as much of the application as possible in the evaluation. It might be easier to fake the context-gathering step of the application and test only the prompt assembly and prompt boilerplate; sometimes, mocking the context is even unavoidable. But often, the context-gathering steps become more important in building a quality LLM application. If you sidestep context gathering or any other aspect of your application, it will be at the peril of application quality assurance, and you might be in for a nasty surprise when the new feature goes into production.

Online Evaluation

With online evaluation, you're looking for user feedback on whether the application provides a good experience. Feedback doesn't have to involve filling out long forms, though. The lifeblood of online evaluation is telemetry data—so measure *everything*.

One obvious way to assess quality is to ask users directly. In ChatGPT and other chat-based LLM experiences, you've probably seen the little thumbs-up or thumbs-down buttons next to each assistant message. While this seems to be a clear metric for quality, you have to account for bias. It might be that only the really angry users ever vote—and they always vote thumbs-down. And besides this, proportionally

speaking, not much traffic gets any interaction with the up/down buttons. So unless your application is really high traffic, you might not get enough data from up/down buttons.

Clearly, we have to get more creative with our measurements—so you must consider *implicit* indicators of quality. For GitHub Copilot code completions, we measure how often completions are accepted and we check to see if users are going back and modifying our completions after accepting them. For your own applications, you'll probably find your own ways of implicitly measuring quality. Be cautious about how you interpret implicit feedback. If you are building an LLM-based scheduling assistant and users are interacting and quickly leaving, then it might be because they are accomplishing their tasks efficiently (Yay!), but it could also be that users are frustrated and are abandoning the experience altogether.

Measure something that matters—something that demonstrates a productivity boost for your customers. Copilot chose the acceptance rate as the key metric because it correlated most highly with the user's productivity gains. For a scheduling assistant, rather than measuring session length, which is ambiguous, look for successfully created calendar events and also keep track of how often users change the details of the events after the fact.

Conclusion

After you learned about how an LLM works in the previous chapters, in this chapter, you learned that the LLM application is effectively a transformation layer between the user's problem domain and the document domain where the LLM does its work. We zoomed in on the feedforward part of the loop, and you learned about how the prompt is formed by collecting context related to the user's problem, extracting the most important parts, and assembling them into the boilerplate text of the prompt document. We then zoomed out and looked at how complex prompt engineering can become as it requires state management, integration with external context, increasingly sophisticated reasoning, and interaction with external tools.

In this chapter, we've touched on every topic in the domain of LLM application development—but only at a very high level. In the next chapters, we'll dig deeply into all of the topics introduced in this chapter. You'll learn more about *where* to pull context from, *how* to create snippets and prioritize them, and *how* to build a prompt that is effective in addressing the user's needs. Then, in later chapters, we'll dig into more advanced applications and go into detail about how you can use these basic concepts to create conversational agency and complicated workflows.

PART II
Core Techniques