CHAPTER 1

Introduction to Prompt Engineering

ChatGPT was released in late November of 2022. By January of the following year, the application had accumulated an estimated 100 million monthly users, making ChatGPT the fastest-growing consumer application *ever*. (In comparison, TikTok took 9 months to reach 100 million users, and Instagram took 2.5 years.) And as you can surely attest, esteemed reader, this public acclaim is well deserved! LLMs—like the one that backs ChatGPT—are revolutionizing the way we work. Rather than running to Google to find answers via a traditional web search, you can easily just ask an LLM to talk about a topic. Rather than reading Stack Overflow or rummaging through blog posts to answer technical questions, you can ask an LLM to write you a personalized tutorial on your exact problem space and then follow it up with a set of questions and answers (a Q&A) about the topic. Rather than following the traditional steps to build a programming library, you can boost your progress by pairing with an LLM-based assistant to build the scaffolding and autocomplete your code as you write it!

And to you, *future* reader, will you use LLMs in ways that we, your humble authors from the year 2024, cannot fathom? If the current trends continue, you'll likely have conversations with LLMs many times during the course of a typical day—in the voice of the IT support assistant when your cable goes out, in a friendly conversation with the corner ATM, and, yes, even with a frustratingly realistic robo dialer. There will be other interactions as well. LLMs will curate your news for you, summarizing the headline stories that you're most likely to be interested in and removing (or perhaps *adding*) biased commentary. You'll use LLMs to assist in your communications by writing and summarizing emails, and office and home assistants will even reach out into the real world and interact on your behalf. In a single day, your personal AI assistant might at one point act as a travel agent, helping you make travel plans, book flights, and reserve hotels; and then at another point, act as a shopping assistant, helping you find and purchase items you need.

Why are LLMs so amazing? It's because they are magic! As futurist Arthur C. Clarke famously stated, "Any sufficiently advanced technology is indistinguishable from magic." We think a machine that you can have a conversation with certainly qualifies as magic, but it's the goal of this book to dispel this magic. We will demonstrate that no matter how uncanny, intuitive, and humanlike LLMs sometimes seem to be, at the core, LLMs are simply models that predict the next word in a block of text—that's it and nothing more! As such, LLMs are merely tools for helping users to accomplish some task, and the way that you interact with these tools is by crafting the *prompt*—the block of text—that they are to complete. This is what we call *prompt engineering*. Through this book, we will build up a practical framework for prompt engineering and ultimately for building LLM applications, which *will* be a magical experience for your users.

This chapter sets the background for the journey you are about to take into prompt engineering. But first, let us tell you about how we, your authors, discovered the magic for ourselves.

LLMs Are Magic

Both authors of this book were early research developers for the GitHub Copilot code completion product. Albert was on the founding team, and John appeared on the scene as Albert was moving on to other distant-horizon LLM research projects.

Albert first discovered the magic halfway through 2020. He puts it as follows:

> Every half year or so, during our ideation meetings in the ML-on-code group, someone would bring up the matter of code synthesis. And the answer was always the same: it will be amazing, one day, but that day won't come for another five years at least. It was our cold fusion.
>
> This was true until the first day I laid hands on an early prototype of the LLM that would become OpenAI Codex. Then I saw that the future was now: cold fusion had finally arrived.
>
> It was immediately clear that this model was wholly different from the sorry stabs at code synthesis we had known before. This model wouldn't just have a chance of predicting the next word—it could generate whole statements and whole functions from just the docstring. Functions that worked!
>
> Before we decided what we could build with this model (spoiler: it would eventually become GitHub's Copilot code completion product), we wanted to quantify how good the model really was. So, we crowdsourced a bunch of GitHub engineers and had them come up with self-contained coding tasks. Some of the tasks were comparatively easy—but these were hardcore coders, and many of their tasks were also pretty involved. A good number of the tasks were the kind a junior developer would turn to Google for, but some would push even a senior developer to Stack Overflow. Yet, if we gave the model a few tries, it could solve most of them.

> We knew it then—this was the engine that would usher in a new age of coding. All we had to do was build the right vehicle around it.

For John, the magical moment came a couple years later, in early 2023, when he was kicking the tires on the vehicle and taking it out for a spin. He recounts it as follows:

> I set up a screen recording session and laid out the coding challenge that I planned to tackle: create a function that takes an integer and returns the text version of that number. So, given an input of 10, the output would be "ten," and given an input of 1,004,712, the output would be "one million four thousand seven hundred twelve." It's harder than you might expect, because, thanks to English, weird exceptions abound. The text versions of numbers between 10 and 20—"eleven," "twelve," and the teens—don't follow the same pattern as numbers in any other decade. The tens place digit breaks expected patterns—for example, if 90 is "ninety" and 80 is "eighty," then why isn't 30 "threety" and 20 "twoty?" But the real twist in my coding challenge was that I wanted to implement the solution in a language in which I had zero personal experience—Rust. Was Copilot up to the challenge?
>
> Normally, when learning a new programming language, I would refer to the typical how-tos: How do I create a variable? How do I create a list? How do I iterate over the items in a list? How do I write an if statement? But with Copilot, I started by just writing a docstring:

```
// GOAL: Create a function that prints a string version of any number
   supplied to the function.
// 1 -> "one"
// 2034 -> "two thousand thirty four"
// 11 -> "eleven"
fn
```

> Copilot saw *fn* and jumped in to help:

```
fn number_to_string(number: i32) -> String {
```

> Perfect! I didn't know how to annotate types for the input arguments or return value of functions, but as we continued to work together, I would direct the high-level flow of work via comments like "Split up the input number into groups of three digits," and Copilot would effectively teach me programming constructs. These included things like how to create vectors and assign them to variables, as in `let mut number_string_vec = Vec::new();` and how to make loops, as in `while number > 0 {`.
>
> The experience was great. I was making progress and learning the language without being distracted by constant references to language tutorials—my project was my tutorial. Then, 20 minutes into this experiment, Copilot blew my mind. I typed a comment and started the next control loop that I knew we would need:

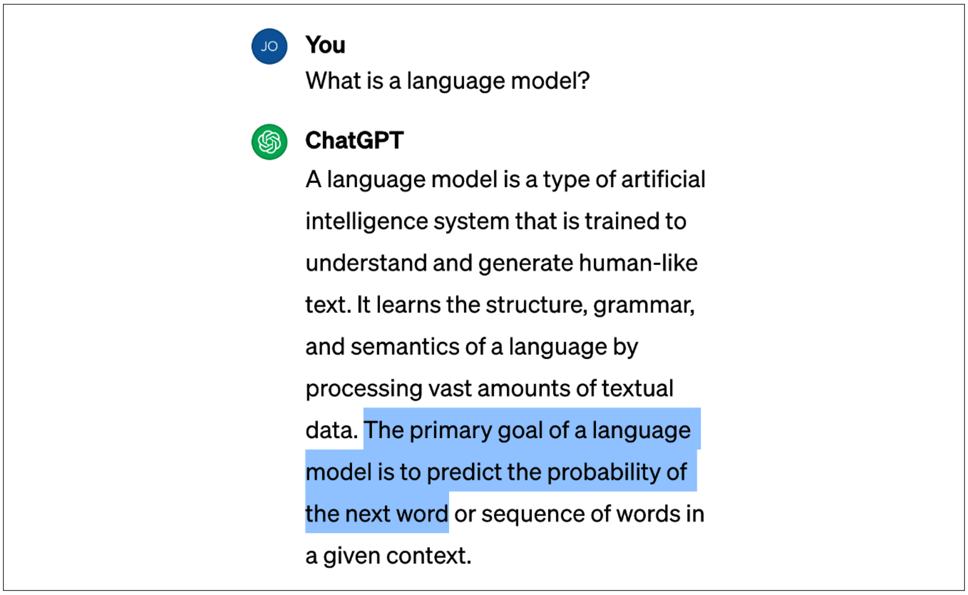
```
// iterate through number_string_vec, assemble the name of the number
// for each order of magnitude, and concatenate to number_string
for
```

After a moment's pause, Copilot interjected 30 lines of code! In the recording, you can actually hear me audibly gasp. The code compiled successfully—it was all syntactically correct—and it ran. The answer was a little wonky. An input of 5,034,012 resulted in the string "five thirty four thousand twelve million," but hey, I wouldn't expect a human to be right the first time, and the bug was easy to spot and correct. By the end of the 40-minute pairing session, I'd done the impossible—I 'd created nontrivial code in a language that I was completely unfamiliar with! Copilot had coached me toward basic understanding of Rust syntax, and it had demonstrated a more abstract grasp of my goals and interjected at several points to help me fill in the details. If I had tried this on my own, I suspect it would have taken hours.

Our magical experiences are not unique. If you're reading this book, you've likely had some mind-blowing interactions with LLMs yourself. Perhaps you first became aware of the power of LLMs with ChatGPT, or maybe your first experience was with one of the first-generation applications that have been pouring out since early 2023: internet search assistants such as Microsoft's Bing or Google's Bard, or document assistants such as Microsoft's broader Copilot suite of tools. But getting to this technological inflection point was not something that happened overnight. To truly understand LLMs, it is important to know how we got here.

Language Models: How Did We Get Here?

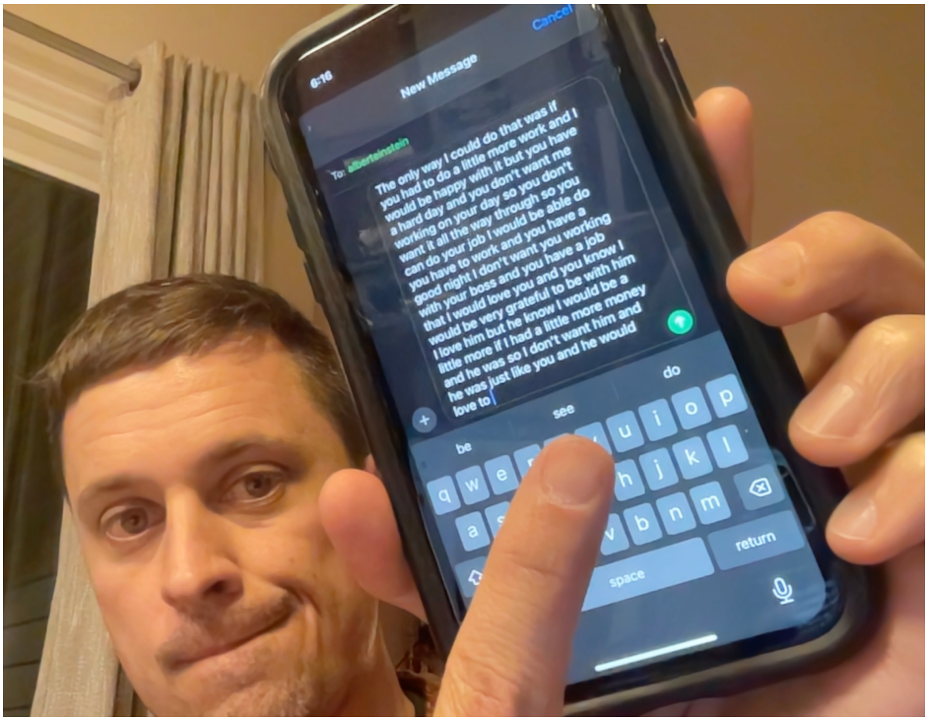
To understand how we got to this very interesting point in the history of technology, we first need to know what a language model actually is and what it does. Who better to ask than the world's most popular LLM application: ChatGPT (see Figure 1-1).


JO **You**
What is a language model?

ChatGPT
A language model is a type of artificial intelligence system that is trained to understand and generate human-like text. It learns the structure, grammar, and semantics of a language by processing vast amounts of textual data. The primary goal of a language model is to predict the probability of the next word or sequence of words in a given context.

Figure 1-1. What is a language model?

See? It's just like we said at the opening of the chapter: the primary goal of a language model is to predict the probability of the next word. You've seen this functionality before, haven't you? It's the bar of completion words that appears above the keypad when you're typing out a text message on your iPhone (see Figure 1-2). You might have never noticed it…*because it isn't that useful.* If this is all that language models do, then how on earth are they currently taking the world by storm?



The only way I could do that was if you had to do a little more work and I would be happy with it but you have a hard day and you don't want me working on your day so you don't want it all the way through so you can do your job I would be able do you have to work and you have a good night I don't want you working with your boss and you have a job that I would love you and you know I would be very grateful to be with him I love him but he know I would be a little more if I had a little more money and he was so I don't want him and he was just like you and he would love to

Figure 1-2. John pointing to the completion bar on his phone

Early Language Models

Language models have actually been around for a long time. If you're reading this book soon after its publication, then the language model that powers the iPhone guess-the-next-word functionality is based upon a Markov model of natural language that was first introduced in 1948. However, there are other more recent language models that have more directly set the stage for the AI revolution that is now underway.

By 2014, the most powerful language models were based on the sequence to sequence (seq2seq) architecture introduced at Google. Seq2seq was a recurrent neural network, which, in theory, should have been ideal for text processing because it processes one token at a time and recurrently updates its internal state. This allows seq2seq to process arbitrarily long sequences of text. With specialized architectures and training, the seq2seq architecture was capable of performing several different types of natural language tasks: classification, entity extraction, translation, summarization, and more. But these models had an Achilles' heel—an information bottleneck limited their capabilities.

The seq2seq architecture has two major components: the encoder and the decoder (see Figure 1-3). Processing starts by sending the encoder a stream of tokens that are processed one at a time. As the tokens are received, the encoder updates a hidden state vector that accumulates information from the input sequence. When the last token has been processed, the final value of the hidden state, called the thought vector, is sent to the decoder. The decoder then uses the information from the thought vector to generate output tokens. The problem, though, is that the thought vector is fixed and finite. It often "forgets" important information from longer blocks of text, giving the decoder little to work with—this is the information bottleneck.

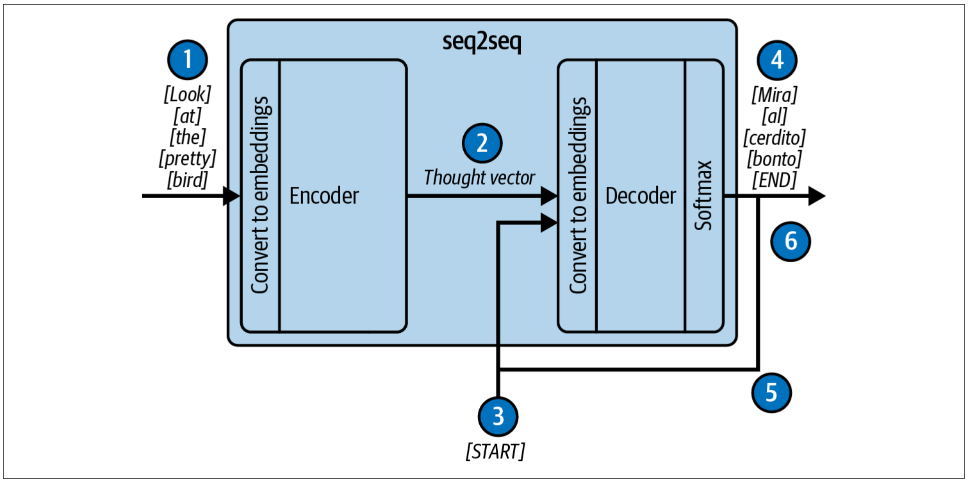


Figure 1-3. A translation seq2seq model

The model in the figure works as follows:

1. Tokens from the source language are sent to the encoder one at a time and converted to an embedding vector, and they update the internal state of the encoder.
2. The internal state is packaged up as the thought vector and sent to the decoder.

3. A special "start" token is sent to the decoder, indicating that this is the start of the output tokens.
4. Conditioned upon the value of the thought vector, the decoder state is updated and an output token from the target language is emitted.
5. The output token is provided as the next input into the decoder. At this point, the process recurrently loops back and forth from step 4 to step 5.
6. Finally, the decoder emits a special "end" token, indicating that the decoding process is complete. The limited thought vector could transfer only a limited amount of information to the decoder.

A 2015 paper, "Neural Machine Translation by Jointly Learning to Align and Translate", introduced a new approach to addressing this bottleneck. Rather than having the encoder supply a single thought vector, it preserved all the hidden state vectors generated for each token encountered in the encoding process and then allowed the decoder to "soft search" over all of the vectors. As a demonstration, the paper showed that using soft search with an English-to-French translation model increased translation quality significantly. This soft search technique soon came to be known as the attention mechanism.

The attention mechanism soon gained a good deal of attention of its own in the AI community, culminating in the 2017 Google Research paper "Attention Is All You Need", which introduced the transformer architecture shown in Figure 1-4. The transformer retained the high-level structure of its predecessor—consisting of an encoder that received tokens as input followed by a decoder that generated output tokens. But unlike the seq2seq model, all of the recurrent circuitry had been removed, and the transformer instead relies completely upon the attention mechanism. The resulting architecture was very flexible and much better at modeling training data than seq2seq. But whereas seq2seq could process arbitrarily long sequences, the transformer could process only a fixed, finite sequence of inputs and outputs. Since the transformer is the direct progenitor of the GPT models, this is a limitation that we have been pushing back against ever since.

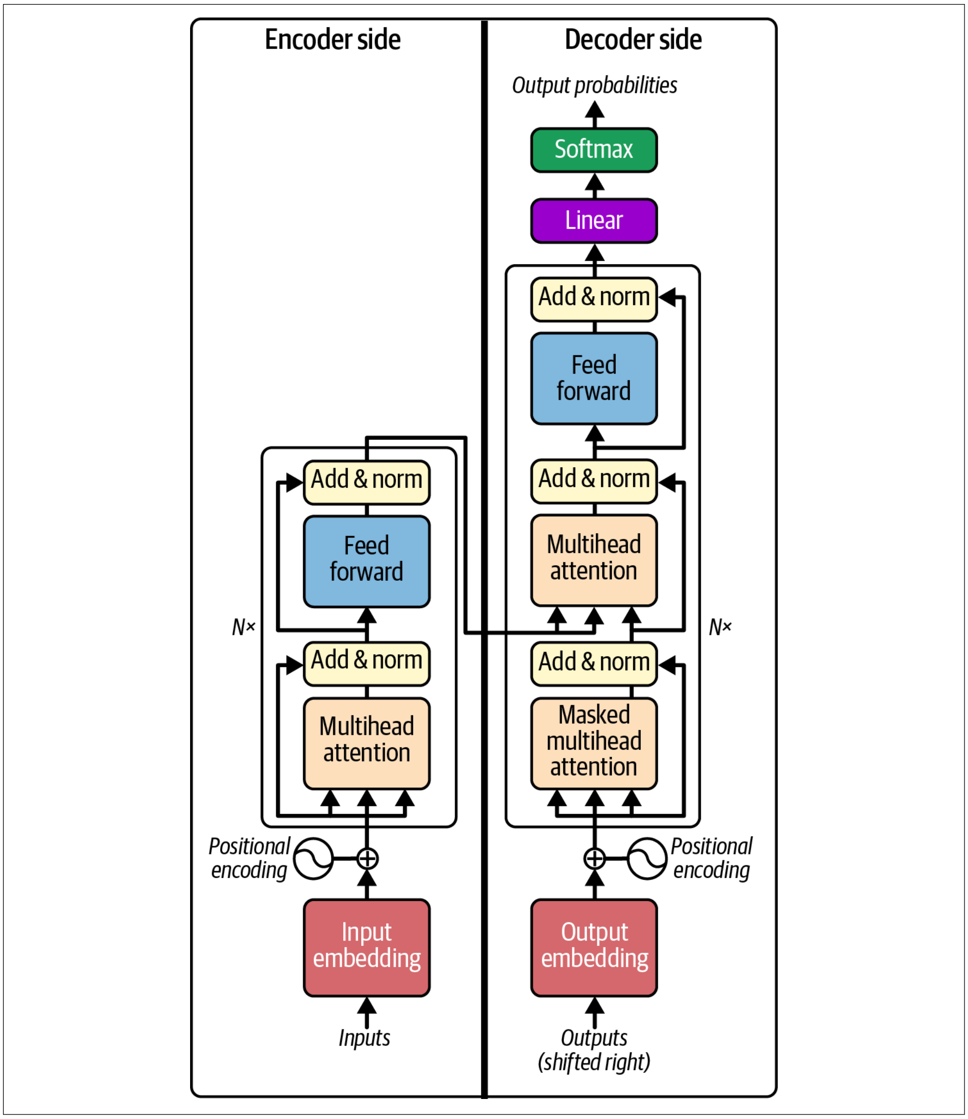


Figure 1-4. Transformer architecture

GPT Enters the Scene

The generative pre-trained transformer architecture was introduced in the 2018 paper "Improving Language Understanding by Generative Pre-Training". The architecture wasn't particularly special or new. Actually, the architecture was just a transformer with the encoder ripped off—it was just the decoder side. However, this simplification led to some unexpected new possibilities that would only be fully realized in coming years. It was this generative pre-trained transformer architecture—GPT—that would soon ignite the ongoing AI revolution.

In 2018, this wasn't apparent. At that point in time, it was standard practice to *pre-train* models with unlabeled data—for instance, scraps of text from the internet—and then modify the architecture of the models and apply specialized fine-tuning so that the final model would then be able to do *one* task very well. And so it was with the generative *pre-trained* transformer architecture. The 2018 paper simply showed that this pattern worked really well for GPTs—pre-training on unlabeled text followed by supervised fine-tuning for a particular task led to really good models for a variety of tasks such as classification, measuring similarities among documents, and answering multiple-choice questions. But we should emphasize one point: after the GPT was fine-tuned, it was only good at the single task for which it was fine-tuned.

GPT-2 was simply a scaled-up version of GPT. When it was introduced in 2019, it was beginning to dawn upon researchers that the GPT architecture was something special. This is clearly evidenced in the second paragraph of the OpenAI blog post introducing GPT-2:

> Our model, called GPT-2 (a successor to GPT), was trained simply to predict the next word in 40 GB of Internet text. Due to our concerns about malicious applications of the technology, we are not releasing the trained model.

Wow! How can those two sentences belong next to each other? How does something as innocuous as predicting the next word—just like an iPhone does when you write a text message—lead to such grave concerns about misuse? If you read the corresponding academic paper, "Language Models Are Unsupervised Multitask Learners", then you start to find out. GPT-2 was 1.5 billion parameters, as compared with GPT's 117 million, and was trained on 40 GB of text, as compared with GPT's 4.5 GB. A simple order-of-magnitude increase in model and training set size led to an unprecedented emergent quality—instead of having to fine-tune GPT-2 for a single task, you could apply the raw, pre-trained model to the task and often achieve better results than state-of-the-art models that were fine-tuned specifically for the task. This included benchmarks for understanding ambiguous pronouns, predicting missing words in text, tagging parts of speech, and more. And despite falling behind the state of the art, GPT-2 also fared surprisingly well on reading comprehension, summarization, translation, and question-answering tasks, again against models fine-tuned specifically for those tasks.

But, why all the concern about "malicious applications" of this model? It's because the model had become quite good at mimicking natural text. And, as the OpenAI blog post indicates, this capability could be used to "generate misleading news articles, impersonate others online, automate the production of abusive or faked content to post on social media, and automate the production of spam/phishing content." If anything, this possibility has only become more real and concerning today than it was in 2019.

GPT-3 saw another order-of-magnitude increase in both model size and training data, with a corresponding leap in capability. The 2020 paper "Language Models Are Few-Shot Learners" showed that, given a few examples of the task you want the model to complete, (a.k.a. "few-shot examples"), the model could faithfully reproduce the input pattern and, as a result, perform just about any language-based task that you could imagine—and often with remarkably high-quality results. This is when we found out that you could modify the input—the prompt—and thereby condition the model to perform the requisite task at hand. This was the birth of prompt engineering.

ChatGPT, released in November 2022, was backed by GPT-3.5—and the rest is history! But, it's a history rapidly in the making (see Table 1-1). In March of 2023, GPT-4 was released, and although the details were not officially revealed, that model was rumored to be another order of magnitude larger in both model size and amount of training data, and it was again much more capable than its predecessors. Since then, more and more models have appeared. Some are from OpenAI while others are from major industry players, such as Llama from Meta, Claude from Anthropic, and Gemini from Google. We have continued to see leaps in quality, and increasingly, the same level of quality is available in smaller and faster models. If anything, *the progress is only accelerating*.

Table 1-1. Details of the GPT-series models, showing the exponential nature of increase in all metrics

Model	Release date	Parameter count	Training data	Training cost
GPT-1	June 11, 2018	117 million	BookCorpus: 4.5 GB of text from 7,000 unpublished books of various genres	1.7e19 FLOP
GPT-2	February 14, 2019 (initial); November 5, 2019 (full)	1.5 billion	WebText: 40 GB of text and 8 million documents from 45 million web pages upvoted on Reddit	1.5e21 FLOP
GPT-3	May 28, 2020	175 billion	499 billion tokens consisting of Common Crawl (570 GB), WebText, English Wikipedia, and two books corpora (Books1 and Books2)	3.1e23 FLOP
GPT-3.5	March 15, 2022	175 billion	Undisclosed	Undisclosed
GPT-4	March 14, 2023	1.8 trillion (rumored)	Rumored to be 13 trillion tokens	Estimated to be 2.1e25 FLOP

Prompt Engineering

Now, we arrive at the beginning of *your* journey into the world of prompt engineering. At their core, LLMs are capable of one thing—completing text. The input into the model is called the *prompt*—it is a document, or block of text, that we expect the model to complete. *Prompt engineering*, then, in its simplest form, is the practice of crafting the prompt so that its completion contains the information required to address the problem at hand.

In this book, we provide a much larger picture of prompt engineering that involves moves well beyond a single prompt and discuss the entire LLM-based application, where prompt construction and the interpretation of the answer are done programmatically. To build a quality piece of software and a quality UX, the prompt engineer must create a pattern for iterative communication among the user, the application, and the LLM. The user conveys their problem to the application, the application constructs a pseudodocument to be sent to the LLM, the LLM completes the document, and finally, the application parses the completion and conveys the result back to the user or otherwise performs an action on the user's behalf. The science *and art* of prompt engineering is to make sure that this communication is structured in a way that best translates among very different domains, the user's problem space, and the document space of LLMs.

Prompt engineering comes in several levels of sophistication. The most basic form makes use of only a very thin application layer. For instance, when you engage with ChatGPT, you're crafting a prompt almost directly; the application is merely wrapping the conversation thread in a special ChatML markdown. (You'll learn more about this in Chapter 3.) Similarly, when GitHub Copilot was first created for code completions, it was doing little more than passing the current file along to the model to complete.

At the next level of sophistication, prompt engineering involves modifying and augmenting the user's input into the model. For instance, LLMs deal with text, so a tech support hotline could transcribe a user's speech to text and use it in the prompt sent to the LLM. Additionally, relevant content from previous help transcripts or from relevant support documentation could be included in the prompt. As a real-world example, as GitHub Copilot code completions developed, we realized that the completion quality improved considerably if we incorporated relevant snippets from the user's neighboring tabs. This makes sense, right? The user had the tabs open because they were referencing information there, so it stands to reason that the model could benefit from this information as well. Another example is the new Bing chat-based search experience. In this instance, content from traditional search results is pulled into the prompt. This allows the assistant to competently discuss information that it never saw in the training data (for instance, because it referred to events that happened after the model was trained). More importantly, this approach helps Bing reduce hallucinations, a topic we'll revisit several times throughout the book, starting in the next chapter.

Another aspect of prompt engineering at this level of sophistication comes when the interactions with the LLM become *stateful*, meaning they maintain context and information from prior interactions. A chat application is the quintessential example here. With each new exchange from the user, the application must recall what happened in previous exchanges and generate a prompt that faithfully represents the interaction. As the conversation or history gets longer, you will have to be careful to not overfill the prompt or include spurious content that might distract the model. You may choose to drop the earliest exchanges or less relevant content from previous exchanges, and you may even employ summarization to compress the content.

Yet another aspect of prompt engineering at this level of sophistication involves giving the LLM-based application tools that allow the LLM to reach out into the real world by making API requests to read information or to even create or modify assets that are available on the internet. For instance, an LLM-based email application might receive this input from a user: "Send Diane an invitation to a meeting on May 5." This application would use one tool to identify Diane in the user's contacts list and then use a calendar API to look up her availability before finally sending an email invitation. As these models get cheaper and more powerful, just imagine the possibilities available with the APIs already at our disposal today! Prompt engineering here is critical. How will the model know which tool to use? How will it use the tool in the correct way? How will your application properly share the information from the tool execution with the model? What do we do when the tool usage results in some sort of error state? We will talk about all of this in Chapter 8.

The final level of sophistication that we cover in this book is how to provide the LLM application with agency—the ability to make its own decisions about how to accomplish broad goals supplied by the user. This is clearly on the frontier of our capabilities with LLMs, but research and practical exploration are underway. Already, you can download AutoGPT and supply it with a goal, and it will take off on a multistep process to gather the information it needs to accomplish the goal. Does it always work? No. Actually, unless the goal is quite constrained, it tends to fail at the task more often than it succeeds. But giving LLM applications some form of agency and autonomy is still an important step toward exciting future possibilities. You'll read our take on this in Chapters 8 and 9.

Conclusion

As we said at the start, this chapter sets the background for the journey you are about to take into prompt engineering. We started with a discussion of the recent history of language models, and we highlighted why LLMs are so special and different—and why they are fueling the AI revolution that we are all now witnessing. We then defined the topic of this book: prompt engineering.

In particular, you should understand that this book isn't going to be all about how to do nitpicky wording of a single prompt to get one good completion. Sure, we'll cover that, and we'll cover in detail all the things you need to do to generate high-quality completions that serve their intended purpose. But when we say, "prompt engineering," we mean building the entire LLM-based application. The LLM application serves as a transformation layer, iteratively and statefully converting real-world needs into text that LLMs can address and then converting the data provided by the LLMs into information and action that address those real-world needs.

Before we set off on this journey, let's make sure we're appropriately packed. In the next chapter, you'll learn how LLM text completion works from the top-level API all the way down to low-level attention mechanisms. In the subsequent chapter, we'll build upon that knowledge to explain how LLMs have been expanded to handle chat and tool usage, and you'll see that deep down, it's really all the same thing—text completion. Then, with those foundational ideas in store, you'll be ready for your journey.