CHAPTER 6

Prompt Engineering

In the first chapters of this book, we took our first steps into the world of large language models (LLMs). We delved into various applications, such as supervised and unsupervised classification, employing models that focus on representing text, like BERT and its derivatives.

As we progressed, we used models trained primarily for text generation, models that are often referred to as *generative pre-trained transformers* (GPT). These models have the remarkable ability to generate text in response to *prompts* from the user. Through *prompt engineering*, we can design these prompts in a way that enhances the quality of the generated text.

In this chapter, we will explore these generative models in more detail and dive into the realm of prompt engineering, reasoning with generative models, verification, and even evaluating their output.

Using Text Generation Models

Before we start with the fundamentals of prompt engineering, it is essential to explore the basics of utilizing a text generation model. How do we select the model to use? Do we use a proprietary or open source model? How can we control the generated output? These questions will serve as our stepping stones into using text generation models.

Choosing a Text Generation Model

Choosing a text generation model starts with choosing between proprietary models or open source models. Although proprietary models are generally more performant, we focus in this book more on open source models as they offer more flexibility and are free to use.

Figure 6-1 shows a small selection of impactful foundation models, LLMs that have been pretrained on vast amounts of text data and are often fine-tuned for specific applications.

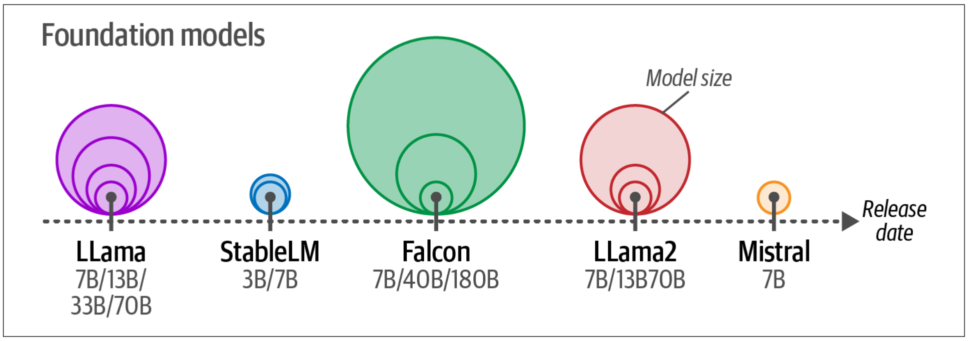


Figure 6-1. Foundation models are often released in several different sizes.

From those foundation models, hundreds if not thousands of models have been fine-tuned, one more suitable for certain tasks than another. Choosing the model to use can be a daunting task!

We advise starting with a small foundation model. So let's continue using Phi-3-mini, which has 3.8 billion parameters. This makes it suitable for running with devices up to 8 GB of VRAM. Overall, scaling up to larger models tends to be a nicer experience than scaling down. Smaller models provide a great introduction and lay a solid foundation for progressing to larger models.

Loading a Text Generation Model

The most straightforward method of loading a model, as we have done in previous chapters, is by leveraging the `transformers` library:

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-instruct")

# Create a pipeline
pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=500,
    do_sample=False,
)
```

Compared to previous chapters, we will take a closer look at developing and using the prompt template.

To illustrate, let's revisit the example from Chapter 1 where we asked the LLM to make a joke about chickens:

```
# Prompt
messages = [
    {"role": "user", "content": "Create a funny joke about chickens."}
]

# Generate the output
output = pipe(messages)
print(output[0]["generated_text"])
```

```
Why don't chickens like to go to the gym? Because they can't crack the egg-
sistence of it!
```

Under the hood, `transformers.pipeline` first converts our messages into a specific prompt template. We can explore this process by accessing the underlying tokenizer:

```
# Apply prompt template
prompt = pipe.tokenizer.apply_chat_template(messages, tokenize=False)
print(prompt)
```

```
<s><|user|>
Create a funny joke about chickens.<|end|>
<|assistant|>
```

You may recognize the special tokens `<|user|>` and `<|assistant|>` from Chapter 2. This prompt template, further illustrated in Figure 6-2, was used during the training of the model. Not only does it provide information about who said what, but it is also used to indicate when the model should stop generating text (see the `<|end|>` token). This prompt is passed directly to the LLM and processed all at once.

In the next chapter, we will customize parts of this template ourselves. Throughout this chapter, we can use `transformers.pipeline` to handle chat template processing for us. Next, let us explore how we can control the output of the model.

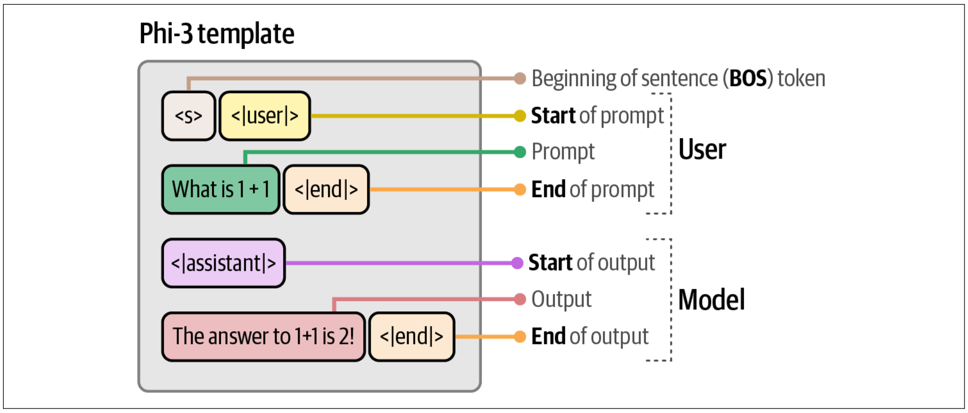


Figure 6-2. The template Phi-3 expects when interacting with the model.

Controlling Model Output

Other than prompt engineering, we can control the kind of output we want by adjusting the model parameters. In our previous example, you might have noticed that we used several parameters in the `pipe` function, including `temperature` and `top_p`.

These parameters control the randomness of the output. A part of what makes LLMs exciting technology is that it can generate different responses for the exact same prompt. Each time an LLM needs to generate a token, it assigns a likelihood number to each possible token.

As illustrated in Figure 6-3, in the sentence "I am driving a…" the likelihood of that sentence being followed by tokens like "car" or "truck" is generally higher than a token like "elephant." However, there is still a possibility of "elephant" being generated but it is much lower.

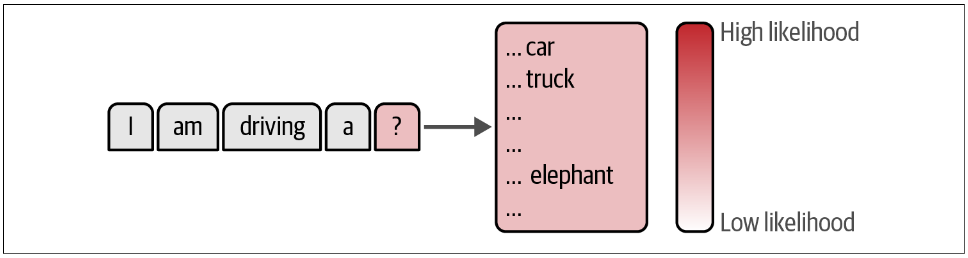


Figure 6-3. The model chooses the next token to generate based on their likelihood scores.

When we loaded our model, we purposefully set `do_sample=False` to make sure the output is somewhat consistent. This means that no sampling will be done and only the most probable next token is selected. However, to use the temperature and top_p parameters, we will set do_sample=True in order to make use of them.

Temperature

The temperature controls the randomness or creativity of the text generated. It defines how likely it is to choose tokens that are less probable. The underlying idea is that a temperature of 0 generates the same response every time because it always chooses the most likely word. As illustrated in Figure 6-4, a higher value allows less probable words to be generated.

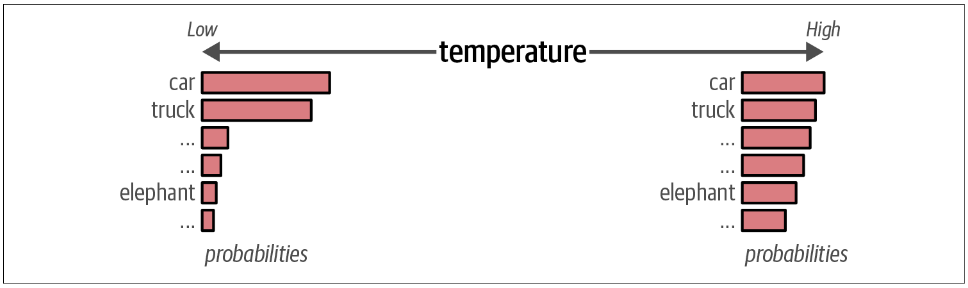


Figure 6-4. A higher temperature increases the likelihood that less probable tokens are generated and vice versa.

As a result, a higher temperature (e.g., 0.8) generally results in a more diverse output while a lower temperature (e.g., 0.2) creates a more deterministic output.

You can use temperature in your pipeline as follows:

```
# Using a high temperature
output = pipe(messages, do_sample=True, temperature=1)
print(output[0]["generated_text"])
```

```
Why don't chickens like to go on a rollercoaster? Because they're afraid they
might suddenly become chicken-soup!
```

Note that every time you rerun this piece of code, the output will change! temperature introduces stochastic behavior since the model now randomly selects tokens.

top_p

top_p, also known as nucleus sampling, is a sampling technique that controls which subset of tokens (the nucleus) the LLM can consider. It will consider tokens until it reaches their cumulative probability. If we set top_p to 0.1, it will consider tokens until it reaches that value. If we set top_p to 1, it will consider all tokens.

As shown in Figure 6-5, by lowering the value, it will consider fewer tokens and generally give less "creative" output, while increasing the value allows the LLM to choose from more tokens.

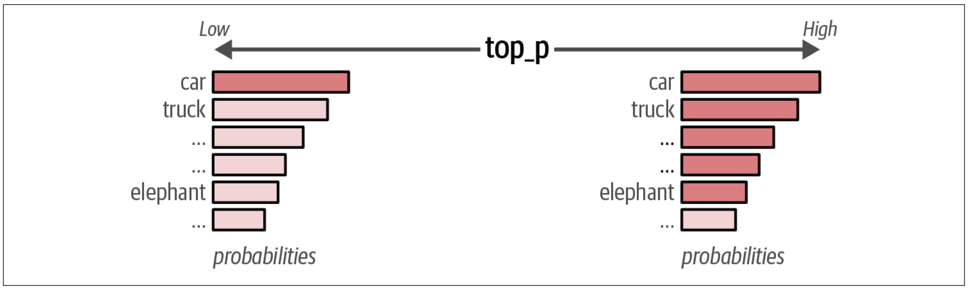


Figure 6-5. A higher `top_p` increases the number of tokens that can be selected to generate and vice versa.

Similarly, the `top_k` parameter controls exactly how many tokens the LLM can consider. If you change its value to 100, the LLM will only consider the top 100 most probable tokens.

You can use `top_p` in your pipeline as follows:

```
# Using a high top_p
output = pipe(messages, do_sample=True, top_p=1)
print(output[0]["generated_text"])
```

```
Why don't chickens make good comedians? Because their 'jokes' always 'feather'
the truth!
```

As shown in Table 6-1, these parameters allow the user to have a sliding scale between being creative (high `temperature` and `top_p`) and being predictable (lower `temperature` and `top_p`).

Table 6-1. Use case examples when selecting values for `temperature` and `top_p`.

Example use case	`Temperature`	`top_p`	Description
Brainstorming session	High	High	High randomness with large pool of potential tokens. The results will be highly diverse, often leading to very creative and unexpected results.
Email generation	Low	Low	Deterministic output with high probable predicted tokens. This results in predictable, focused, and conservative outputs.
Creative writing	High	Low	High randomness with a small pool of potential tokens. This combination produces creative outputs but still remains coherent.
Translation	Low	High	Deterministic output with high probable predicted tokens. Produces coherent output with a wider range of vocabulary, leading to outputs with linguistic variety.

Intro to Prompt Engineering

An essential part of working with text-generative LLMs is prompt engineering. By carefully designing our prompts we can guide the LLM to generate desired responses. Whether the prompts are questions, statements, or instructions, the main goal of prompt engineering is to elicit a useful response from the model.

Prompt engineering is more than designing effective prompts. It can be used as a tool to evaluate the output of a model as well as to design safeguards and safety mitigation methods. This is an iterative process of prompt optimization and requires experimentation. There is not and unlikely will ever be a perfect prompt design.

In this section, we will go through common methods for prompt engineering, and small tips and tricks to understand what the effect is of certain prompts. These skills allow us to understand the capabilities of LLMs and lie at the foundation of interfacing with these kinds of models.

We begin by answering the question: what should be in a prompt?

The Basic Ingredients of a Prompt

An LLM is a prediction machine. Based on a certain input, the prompt, it tries to predict the words that might follow it. At its core (illustrated in Figure 6-6), the prompt does not need to be more than a few words to elicit a response from the LLM.

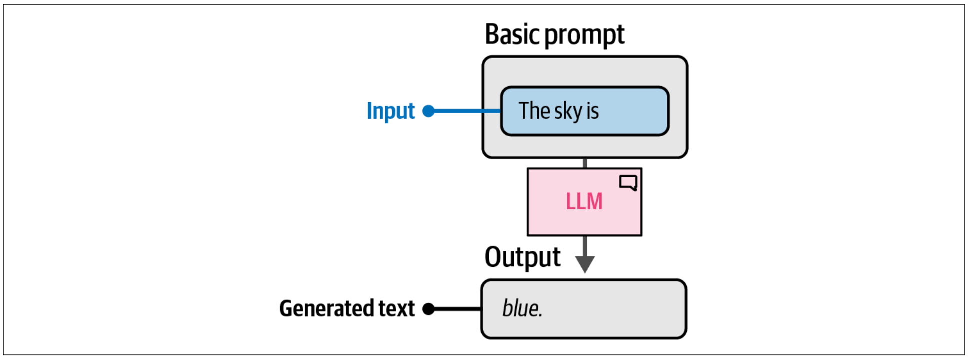


Figure 6-6. A basic example of a prompt. No instruction is given so the LLM will simply try to complete the sentence.

However, although the illustration works as a basic example, it fails to complete a specific task. Instead, we generally approach prompt engineering by asking a specific question or task the LLM should complete. To elicit the desired response, we need a more structured prompt.

For example, and as shown in Figure 6-7, we could ask the LLM to classify a sentence into either having positive or negative sentiment. This extends the most basic prompt to one consisting of two components—the instruction itself and the data that relates to the instruction.

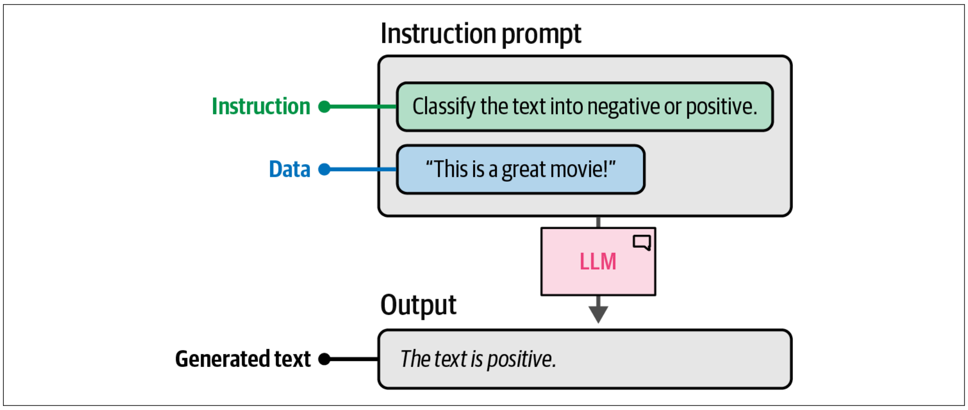


Figure 6-7. Two components of a basic instruction prompt: the instruction itself and the data it refers to.

More complex use cases might require more components in a prompt. For instance, to make sure the model only outputs "negative" or "positive" we can introduce output indicators that help guide the model. In Figure 6-8, we prefix the sentence with "Text:" and add "Sentiment:" to prevent the model from generating a complete sentence. Instead, this structure indicates that we expect either "negative" or "positive." Although the model might not have been trained on these components directly, it was fed enough instructions to be able to generalize to this structure.

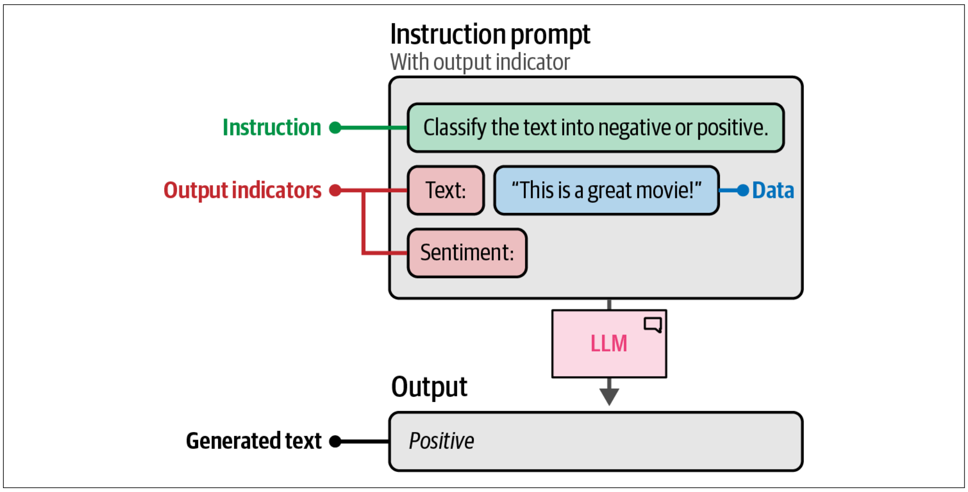


Figure 6-8. Extending the prompt with an output indicator that allows for a specific output.

We can continue adding or updating the elements of a prompt until we elicit the response we are looking for. We could add additional examples, describe the use case in more detail, provide additional context, etc. These components are merely examples and not a limited set of possibilities. The creativity that comes with designing these components is key.

Although a prompt is a single piece of text, it is tremendously helpful to think of prompts as pieces of a larger puzzle. Have I described the context of my question? Does the prompt have an example of the output?

Instruction-Based Prompting

Although prompting comes in many flavors, from discussing philosophy with the LLM to role-playing with your favorite superhero, prompting is often used to have the LLM answer a specific question or resolve a certain task. This is referred to as *instruction-based prompting*.

Figure 6-9 illustrates a number of use cases in which instruction-based prompting plays an important role. We already did one of these in the previous example, namely supervised classification.

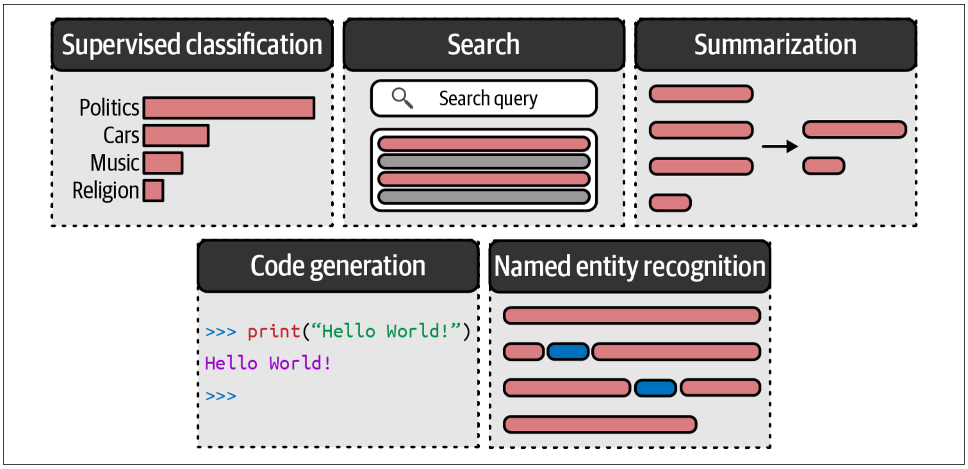


Figure 6-9. Use cases for instruction-based prompting.

Each of these tasks requires different prompting formats and more specifically, asking different questions of the LLM. Asking the LLM to summarize a piece of text will not suddenly result in classification. To illustrate, examples of prompts for some of these use cases can be found in Figure 6-10.

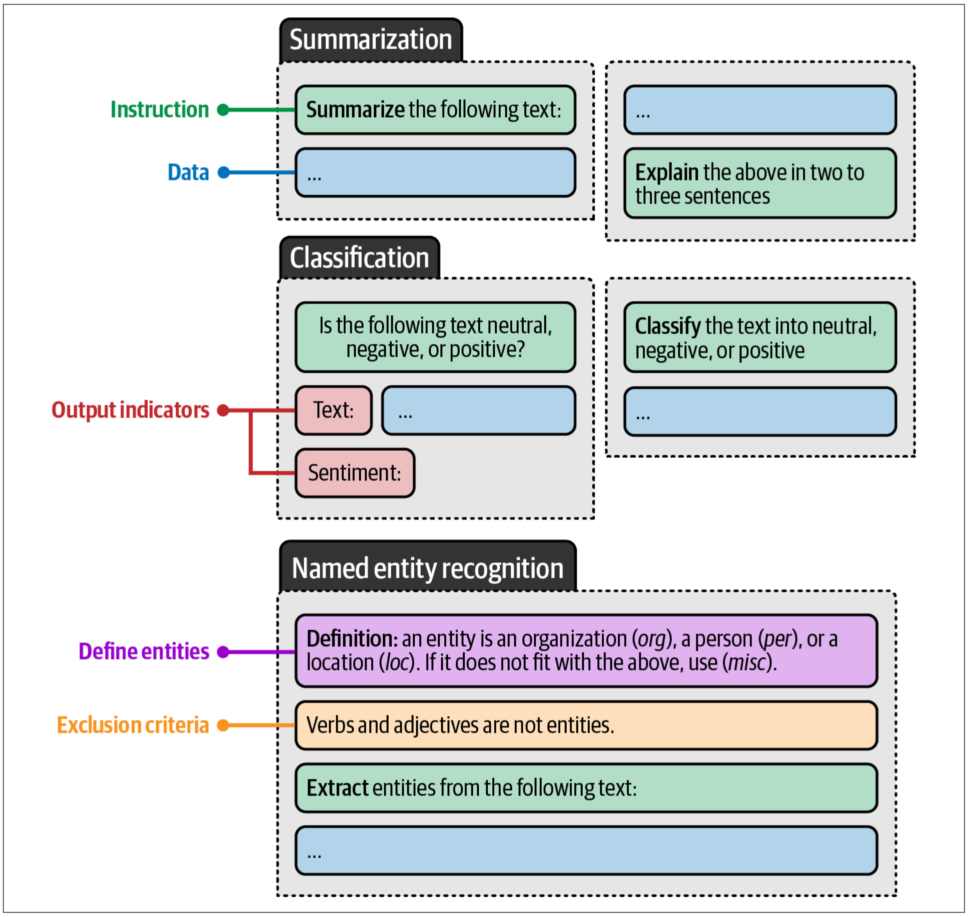

Summarization
Instruction
Data
Summarize the following text:
...
...
Explain the above in two to three sentences
Classification
Is the following text neutral, negative, or positive?
Output indicators
Text:
...
Sentiment:
Classify the text into neutral, negative, or positive
...
Named entity recognition
Define entities
Definition: an entity is an organization (org), a person (per), or a location (loc). If it does not fit with the above, use (misc).
Exclusion criteria
Verbs and adjectives are not entities.
Extract entities from the following text:
...

Figure 6-10. Prompt examples of common use cases. Notice how within a use case, the structure and location of the instruction can be changed.

Although these tasks require different instructions, there is actually a lot of overlap in the prompting techniques used to improve the quality of the output. A non-exhaustive list of these techniques includes:

Specificity
: Accurately describe what you want to achieve. Instead of asking the LLM to "Write a description for a product" ask it to "Write a description for a product in less than two sentences and use a formal tone."

Hallucination
: LLMs may generate incorrect information confidently, which is referred to as hallucination. To reduce its impact, we can ask the LLM to only generate an answer if it knows the answer. If it does not know the answer, it can respond with "I don't know."

Order
: Either begin or end your prompt with the instruction. Especially with long prompts, information in the middle is often forgotten.[1] LLMs tend to focus on information either at the beginning of a prompt (primacy effect) or the end of a prompt (recency effect).

Here, specificity is arguably the most important aspect. By restricting and specifying what the model should generate, there is a smaller chance of having it generate something not related to your use case. For instance, if we were to skip the instruction "in two to three sentences" it might generate complete paragraphs. Like human conversations, without any specific instructions or additional context, it is difficult to derive what the task at hand actually is.

Advanced Prompt Engineering

On the surface, creating a good prompt might seem straightforward. Ask a specific question, be accurate, add some examples, and you are done! However, prompting can grow complex quite quickly and as a result is an often-underestimated component of leveraging LLMs.

Here, we will go through several advanced techniques for building up your prompts, starting with the iterative workflow of building up complex prompts all the way to using LLMs sequentially to get improved results. Eventually, we will even build up to advanced reasoning techniques.

The Potential Complexity of a Prompt

As we explored in the intro to prompt engineering, a prompt generally consists of multiple components. In our very first example, our prompt consisted of instruction, data, and output indicators. As we mentioned before, no prompt is limited to just these three components and you can build it up to be as complex as you want.

These advanced components can quickly make a prompt quite complex. Some common components are:

1 Nelson F. Liu et al. "Lost in the middle: How language models use long contexts." *arXiv preprint arXiv:2307.03172* (2023).

Persona
Describe what role the LLM should take on. For example, use “You are an expert in astrophysics” if you want to ask a question about astrophysics.

Instruction
The task itself. Make sure this is as specific as possible. We do not want to leave much room for interpretation.

Context
Additional information describing the context of the problem or task. It answers questions like “What is the reason for the instruction?”

Format
The format the LLM should use to output the generated text. Without it, the LLM will come up with a format itself, which is troublesome in automated systems.

Audience
The target of the generated text. This also describes the level of the generated output. For education purposes, it is often helpful to use ELI5 (“Explain it like I’m 5”).

Tone
The tone of voice the LLM should use in the generated text. If you are writing a formal email to your boss, you might not want to use an informal tone of voice.

Data
The main data related to the task itself.

To illustrate, let us extend the classification prompt we had earlier and use all of the preceding components. This is demonstrated in Figure 6-11.

This complex prompt demonstrates the modular nature of prompting. We can add and remove components freely and judge their effect on the output. As illustrated in Figure 6-12, we can slowly build up our prompt and explore the effect of each change.

The changes are not limited to simply introducing or removing components. Their order, as we saw before with the recency and primacy effects, can affect the quality of the LLM’s output. In other words, experimentation is vital when finding the best prompt for your use case. With prompting, we essentially have ourselves in an iterative cycle of experimentation.

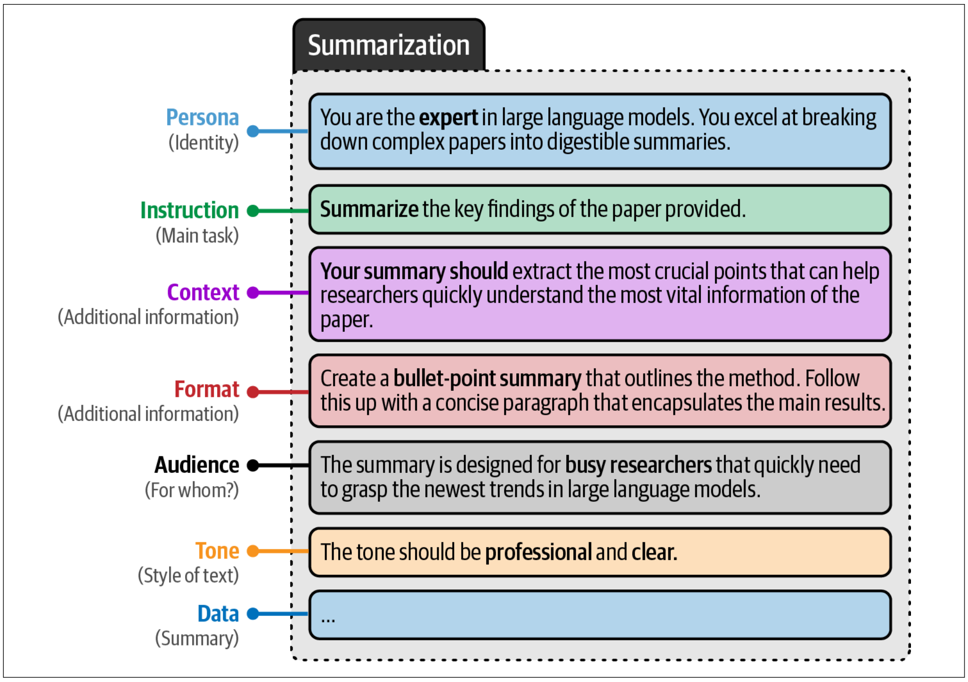

Summarization
Persona (Identity)
You are the expert in large language models. You excel at breaking down complex papers into digestible summaries.
Instruction (Main task)
Summarize the key findings of the paper provided.
Context (Additional information)
Your summary should extract the most crucial points that can help researchers quickly understand the most vital information of the paper.
Format (Additional information)
Create a bullet-point summary that outlines the method. Follow this up with a concise paragraph that encapsulates the main results.
Audience (For whom?)
The summary is designed for busy researchers that quickly need to grasp the newest trends in large language models.
Tone (Style of text)
The tone should be professional and clear.
Data (Summary)
...

Figure 6-11. An example of a complex prompt with many components.

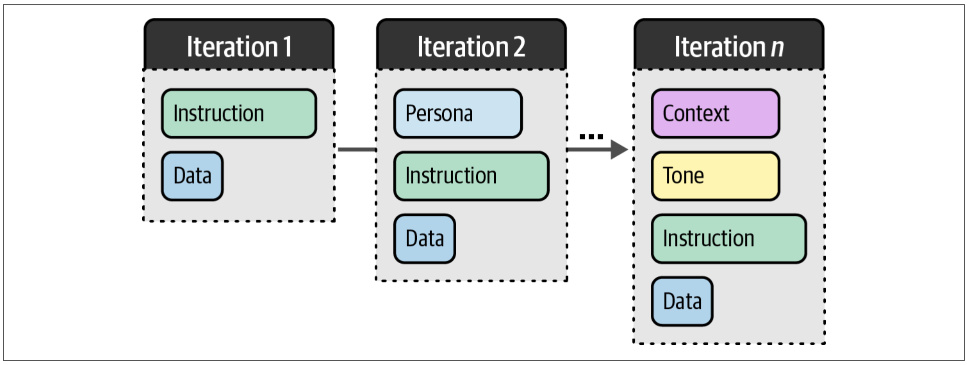


Figure 6-12. Iterating over modular components is a vital part of prompt engineering.

Try it out yourself! Use the complex prompt to add and/or remove parts to observe its impact on the generated output. You will quickly notice when pieces of the puzzle are worth keeping. You can use your own data by adding it to the `data` variable:

```
# Prompt components
persona = "You are an expert in Large Language models. You excel at breaking
down complex papers into digestible summaries.\n"
instruction = "Summarize the key findings of the paper provided.\n"
context = "Your summary should extract the most crucial points that can help
researchers quickly understand the most vital information of the paper.\n"
data_format = "Create a bullet-point summary that outlines the method. Follow
this up with a concise paragraph that encapsulates the main results.\n"
audience = "The summary is designed for busy researchers that quickly need to
grasp the newest trends in Large Language Models.\n"
tone = "The tone should be professional and clear.\n"
text = "MY TEXT TO SUMMARIZE"
data = f"Text to summarize: {text}"

# The full prompt - remove and add pieces to view its impact on the generated
output
query = persona + instruction + context + data_format + audience + tone + data
```

There is all manner of components that we could add and creative components like using emotional stimuli (e.g., "This is very important for my career."[2]). Part of the fun in prompt engineering is that you can be as creative as possible to figure out which combination of prompt components contribute to your use case. There are few constraints to developing a format that works for you.

In a way, it is an attempt to reverse engineer what the model has learned and how it responds to certain prompts. However, note that some prompts work better for certain models compared to others as their training data might be different or they are trained for different purposes.

In-Context Learning: Providing Examples

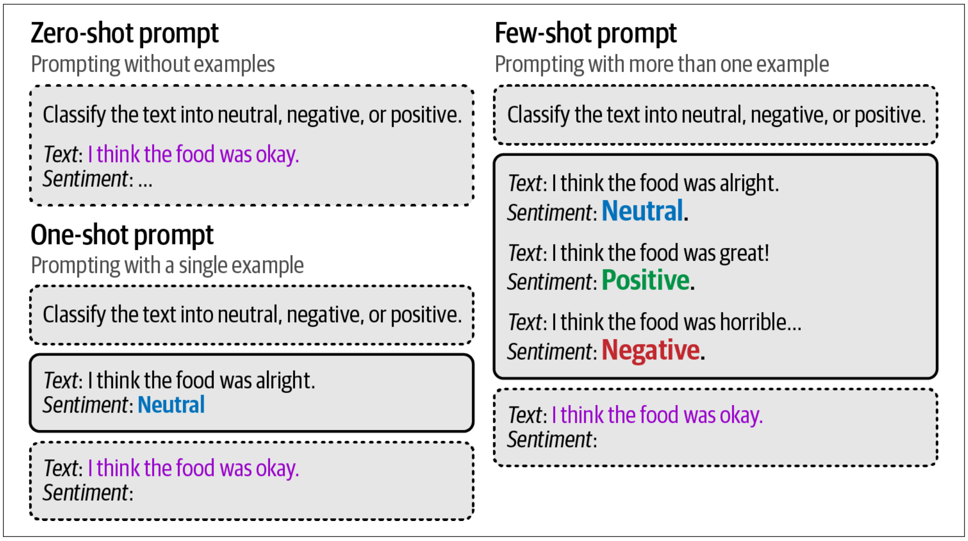
In the previous sections, we tried to accurately describe what the LLM should do. Although accurate and specific descriptions help the LLM to understand the use case, we can go one step further. Instead of describing the task, why do we not just show the task?

We can provide the LLM with examples of exactly the thing that we want to achieve. This is often referred to as *in-context learning*, where we provide the model with correct examples.[3]

2 Cheng Li et al. "EmotionPrompt: Leveraging psychology for large language models enhancement via emotional stimulus." *arXiv preprint arXiv:2307.11760* (2023).

3 Tom Brown et al. "Language models are few-shot learners." *Advances in Neural Information Processing Systems* 33 (2020): 1877–1901.

As illustrated in Figure 6-13, this comes in a number of forms depending on how many examples you show the LLM. Zero-shot prompting does not leverage examples, one-shot prompts use a single example, and few-shot prompts use two or more examples.



Zero-shot prompt
Prompting without examples
Classify the text into neutral, negative, or positive.
Text: I think the food was okay.
Sentiment: ...
One-shot prompt
Prompting with a single example
Classify the text into neutral, negative, or positive.
Text: I think the food was alright.
Sentiment: Neutral
Text: I think the food was okay.
Sentiment:
Few-shot prompt
Prompting with more than one example
Classify the text into neutral, negative, or positive.
Text: I think the food was alright.
Sentiment: Neutral.
Text: I think the food was great!
Sentiment: Positive.
Text: I think the food was horrible...
Sentiment: Negative.
Text: I think the food was okay.
Sentiment:

Figure 6-13. An example of a complex prompt with many components.

Adopting the original phrase, we believe that "an example is worth a thousand words." These examples provide a direct example of what and how the LLM should achieve.

We can illustrate this method with a simple example taken from the original paper describing this method.[4] The goal of the prompt is to generate a sentence with a made-up word. To improve the quality of the resulting sentence, we can show the generative model an example of what a proper sentence with a made-up word would be.

To do so, we will need to differentiate between our question (`user`) and the answers that were provided by the model (`assistant`). We additionally showcase how this interaction is processed using the template:

```
# Use a single example of using the made-up word in a sentence
one_shot_prompt = [
    {
        "role": "user",
        "content": "A 'Gigamuru' is a type of Japanese musical instrument. An
example of a sentence that uses the word Gigamuru is:"
```

4 Ibid.

```
    },
    {
        "role": "assistant",
        "content": "I have a Gigamuru that my uncle gave me as a gift. I love
to play it at home."
    },
    {
        "role": "user",
        "content": "To 'screeg' something is to swing a sword at it. An example
of a sentence that uses the word screeg is:"
    }
]
print(tokenizer.apply_chat_template(one_shot_prompt, tokenize=False))
```

```
<s><|user|>
A 'Gigamuru' is a type of Japanese musical instrument. An example of a sen-
tence that uses the word Gigamuru is:<|end|>
<|assistant|>
I have a Gigamuru that my uncle gave me as a gift. I love to play it at home.<|
end|>
<|user|>
To 'screeg' something is to swing a sword at it. An example of a sentence that
uses the word screeg is:<|end|>
<|assistant|>
```

The prompt illustrates the need to differentiate between the user and the assistant. If we did not, it would seem as if we were talking to ourselves. Using these interactions, we can generate output as follows:

```
# Generate the output
outputs = pipe(one_shot_prompt)
print(outputs[0]["generated_text"])
```

```
During the intense duel, the knight skillfully screeged his opponent's shield,
forcing him to defend himself.
```

It correctly generated the answer!

As with all prompt components, one- or few-shot prompting is not the be all and end all of prompt engineering. We can use it as one piece of the puzzle to further enhance the descriptions that we gave it. The model can still "choose," through random sampling, to ignore the instructions.

Chain Prompting: Breaking up the Problem

In previous examples, we explored splitting up prompts into modular components to improve the performance of LLMs. Although this works well for many use cases, this might not be feasible for highly complex prompts or use cases.

Instead of breaking the problem within a prompt, we can do so between prompts. Essentially, we take the output of one prompt and use it as input for the next, thereby creating a continuous chain of interactions that solves our problem.

To illustrate, let us say we want to use an LLM to create a product name, slogan, and sales pitch for us based on a number of product features. Although we can ask the LLM to do this in one go, we can instead break up the problem into pieces.

As a result, and as illustrated in Figure 6-14, we get a sequential pipeline that first creates the product name, uses that with the product features as input to create the slogan, and finally, uses the features, product name, and slogan to create the sales pitch.

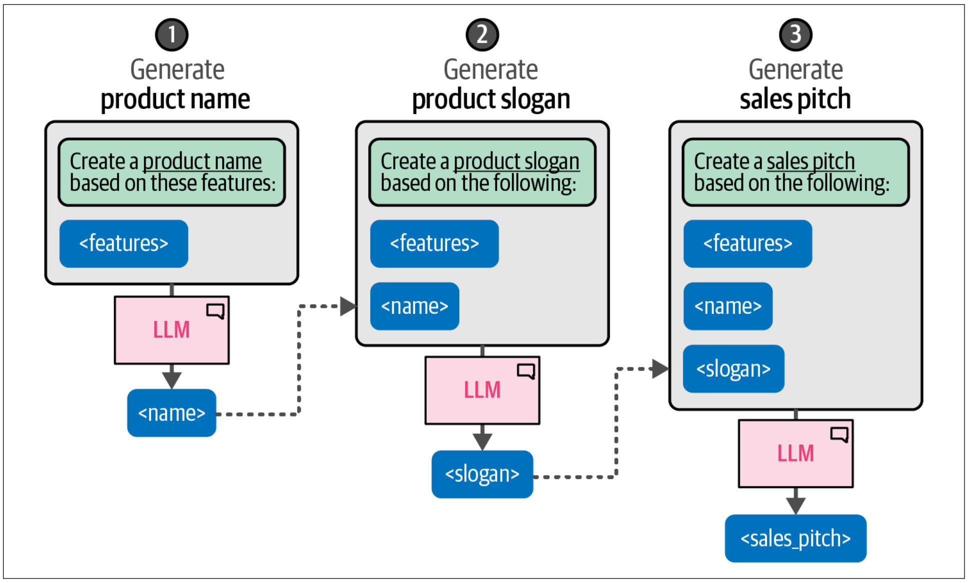


Figure 6-14. Using a description of a product's features, chain prompts to create a suitable name, slogan, and sales pitch.

This technique of chaining prompts allows the LLM to spend more time on each individual question instead of tackling the whole problem. Let us illustrate this with a small example. We first create a name and slogan for a chatbot:

```
# Create name and slogan for a product
product_prompt = [
    {"role": "user", "content": "Create a name and slogan for a chatbot that
leverages LLMs."}
]
outputs = pipe(product_prompt)
product_description = outputs[0]["generated_text"]
print(product_description)
```

```
Name: 'MindMeld Messenger'

Slogan: 'Unleashing Intelligent Conversations, One Response at a Time'
```

Then, we can use the generated output as input for the LLM to generate a sales pitch:

```
# Based on a name and slogan for a product, generate a sales pitch
sales_prompt = [
    {"role": "user", "content": f"Generate a very short sales pitch for the
following product: '{product_description}'"}
]
outputs = pipe(sales_prompt)
sales_pitch = outputs[0]["generated_text"]
print(sales_pitch)
```

```
Introducing MindMeld Messenger - your ultimate communication partner! Unleash
intelligent conversations with our innovative AI-powered messaging platform.
With MindMeld Messenger, every response is thoughtful, personalized, and
timely. Say goodbye to generic replies and hello to meaningful interactions.
Elevate your communication game with MindMeld Messenger - where every message
is a step toward smarter conversations. Try it now and experience the future
of messaging!
```

Although we need two calls to the model, a major benefit is that we can give each call different parameters. For instance, the number of tokens created was relatively small for the name and slogan whereas the pitch can be much longer.

This can be used for a variety of use cases, including:

Response validation
: Ask the LLM to double-check previously generated outputs.

Parallel prompts
: Create multiple prompts in parallel and do a final pass to merge them. For example, ask multiple LLMs to generate multiple recipes in parallel and use the combined result to create a shopping list.

Writing stories
: Leverage the LLM to write books or stories by breaking down the problem into components. For example, by first writing a summary, developing characters, and building the story beats before diving into creating the dialogue.

In the next chapter, we will automate this process and go beyond chaining LLMs. We will chain other pieces of technology together, like memory, tool use, and more! Before that, this idea of prompt chaining will be explored further in the following sections, which describe more complex prompt chaining methods like self-consistency, chain-of-thought, and tree-of-thought.

Reasoning with Generative Models

In the previous sections, we focused mostly on the modular component of prompts, building them up through iteration. These advanced prompt engineering techniques, like prompt chaining, proved to be the first step toward enabling complex reasoning with generative models.

Reasoning is a core component of human intelligence and is often compared to the emergent behavior of LLMs that often resembles reasoning. We highlight "resemble" as these models, at the time of writing, are generally considered to demonstrate this behavior through memorization of training data and pattern matching.

The output that they showcase, however, can demonstrate complex behavior and although it might not be "true" reasoning, they are still referred to as reasoning capabilities. In other words, we work together with the LLM through prompt engineering so we can mimic reasoning processes in order to improve the output of the LLM.

To allow for this reasoning behavior, it is a good moment to step back and explore what reasoning entails in human behavior. To simplify, our methods of reasoning can be divided into system 1 and 2 thinking processes.

System 1 thinking represents an automatic, intuitive, and near-instantaneous process. It shares similarities with generative models that automatically generate tokens without any self-reflective behavior. In contrast, system 2 thinking is a conscious, slow, and logical process, akin to brainstorming and self-reflection.[5]

If we could give a generative model the ability to mimic a form of self-reflection, we would essentially be emulating the system 2 way of thinking, which tends to produce more thoughtful responses than system 1 thinking. In this section, we will explore several techniques that attempt to mimic these kinds of thought processes of human reasoners with the aim of improving the output of the model.

Chain-of-Thought: Think Before Answering

The first and major step toward complex reasoning in generative models was through a method called chain-of-thought. Chain-of-thought aims to have the generative model "think" first rather than answering the question directly without any reasoning.[6]

As illustrated in Figure 6-15, it provides examples in a prompt that demonstrate the reasoning the model should do before generating its response. These reasoning processes are referred to as "thoughts." This helps tremendously for tasks that involve a higher degree of complexity, like mathematical questions. Adding this reasoning step allows the model to distribute more compute over the reasoning process. Instead

5 Daniel Kahneman. *Thinking, Fast and Slow*. Macmillan (2011).

6 Jason Wei et al. "Chain-of-thought prompting elicits reasoning in large language models." *Advances in Neural Information Processing Systems* 35 (2022): 24824–24837.

of calculating the entire solution based on a few tokens, each additional token in this reasoning process allows the LLM to stabilize its output.

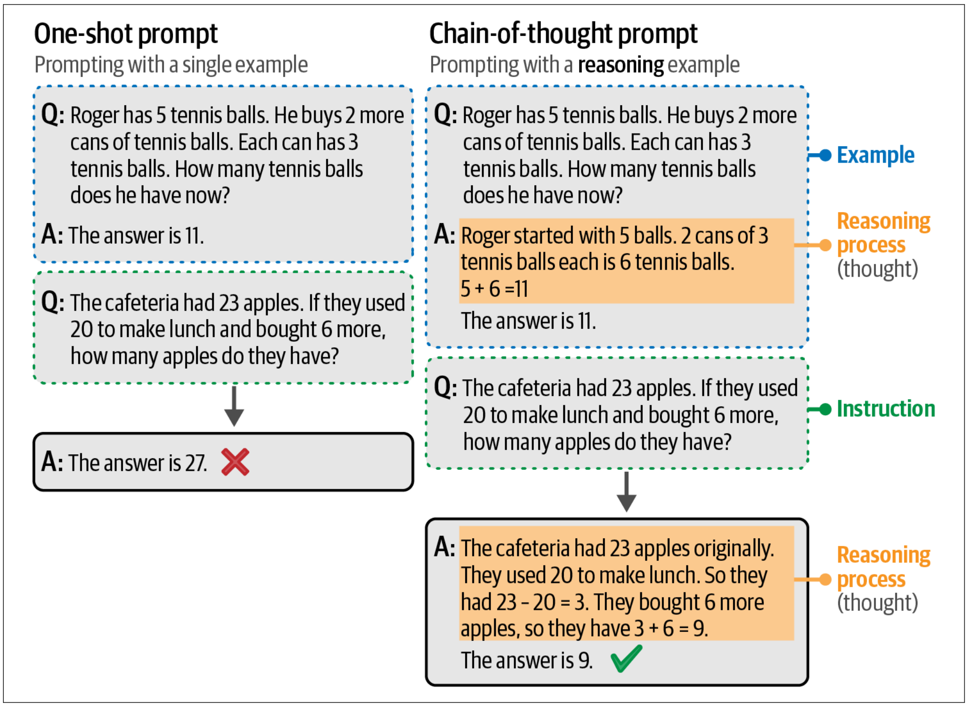

One-shot prompt
Prompting with a single example
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
A: The answer is 27.
Chain-of-thought prompt
Prompting with a reasoning example
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
A: Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 =11
The answer is 11.
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9.
The answer is 9.
Example
Reasoning process (thought)
Instruction
Reasoning process (thought)

Figure 6-15. Chain-of-thought prompting uses reasoning examples to persuade the generative model to use reasoning in its answer.

We use the example the authors used in their paper to demonstrate this phenomenon:

```
# Answering with chain-of-thought
cot_prompt = [
    {"role": "user", "content": "Roger has 5 tennis balls. He buys 2 more cans
of tennis balls. Each can has 3 tennis balls. How many tennis balls does he
have now?"},
    {"role": "assistant", "content": "Roger started with 5 balls. 2 cans of 3
tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer is 11."},
    {"role": "user", "content": "The cafeteria had 23 apples. If they used 20
to make lunch and bought 6 more, how many apples do they have?"}
]

# Generate the output
outputs = pipe(cot_prompt)
print(outputs[0]["generated_text"])
```

```
The cafeteria started with 23 apples. They used 20 apples, so they had 23 - 20
= 3 apples left. Then they bought 6 more apples, so they now have 3 + 6 = 9
apples. The answer is 9.
```

Note how the model doesn't generate only the answer but provides an explanation before doing so. By doing so, it can leverage the knowledge it has generated thus far to compute the final answer.

Although chain-of-thought is a great method for enhancing the output of a generative model, it does require one or more examples of reasoning in the prompt, which the user might not have access to. Instead of providing examples, we can simply ask the generative model to provide the reasoning (zero-shot chain-of-thought). There are many different forms that work but a common and effective method is to use the phrase "Let's think step-by-step," which is illustrated in Figure 6-16.[7]

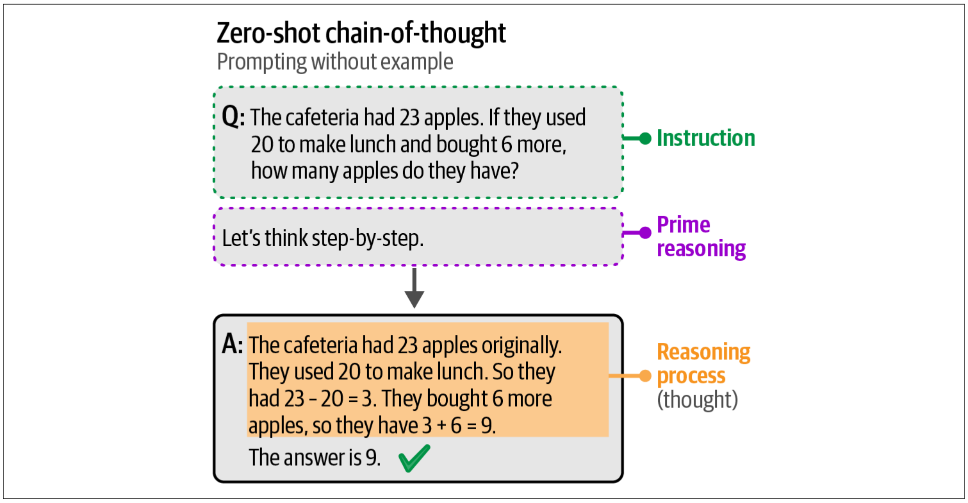

Zero-shot chain-of-thought
Prompting without example
Q: The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more, how many apples do they have?
Instruction
Let's think step-by-step.
Prime reasoning
A: The cafeteria had 23 apples originally. They used 20 to make lunch. So they had 23 - 20 = 3. They bought 6 more apples, so they have 3 + 6 = 9.
The answer is 9.
Reasoning process (thought)

Figure 6-16. Chain-of-thought prompting without using examples. Instead, it uses the phrase "Let's think step-by-step" to prime reasoning in its answer.

Using the example we used before, we can simply append that phrase to the prompt to enable chain-of-thought-like reasoning:

```
# Zero-shot chain-of-thought
zeroshot_cot_prompt = [
    {"role": "user", "content": "The cafeteria had 23 apples. If they used 20
to make lunch and bought 6 more, how many apples do they have? Let's think
step-by-step."}
]

# Generate the output
outputs = pipe(zeroshot_cot_prompt)
print(outputs[0]["generated_text"])
```

7 Takeshi Kojima et al. "Large language models are zero-shot reasoners." *Advances in Neural Information Processing Systems* 35 (2022): 22199–22213.

```
Step 1: Start with the initial number of apples, which is 23.
Step 2: Subtract the number of apples used to make lunch, which is 20. So, 23
- 20 = 3 apples remaining.
Step 3: Add the number of apples bought, which is 6. So, 3 + 6 = 9 apples.

The cafeteria now has 9 apples.
```

Without needing to provide examples, we again got the same reasoning behavior. This is why it is so important to "show your work" when doing calculations. By addressing the reasoning process the LLM can use the previously generated information as a guide through generating the final answer.

Although the prompt "Let's think step by step" can improve the output, you are not constrained by this exact formulation. Alternatives exist like "Take a deep breath and think step-by-step" and "Let's work through this problem step-by-step."[8]

Self-Consistency: Sampling Outputs

Using the same prompt multiple times can lead to different results if we allow for a degree of creativity through parameters like `temperature` and `top_p`. As a result, the quality of the output might improve or degrade depending on the random selection of tokens. In other words, luck!

To counteract this degree of randomness and improve the performance of generative models, self-consistency was introduced. This method asks the generative model the same prompt multiple times and takes the majority result as the final answer.[9] During this process, each answer can be affected by different `temperature` and `top_p` values to increase the diversity of sampling.

As illustrated in Figure 6-17, this method can further be improved by adding chain-of-thought prompting to improve its reasoning while only using the answer for the voting procedure.

8 Chengrun Yang et al. "Large language models as optimizers." *arXiv preprint arXiv:2309.03409* (2023).

9 Xuezhi Wang et al. "Self-consistency improves chain of thought reasoning in language models." *arXiv preprint arXiv:2203.11171* (2022).

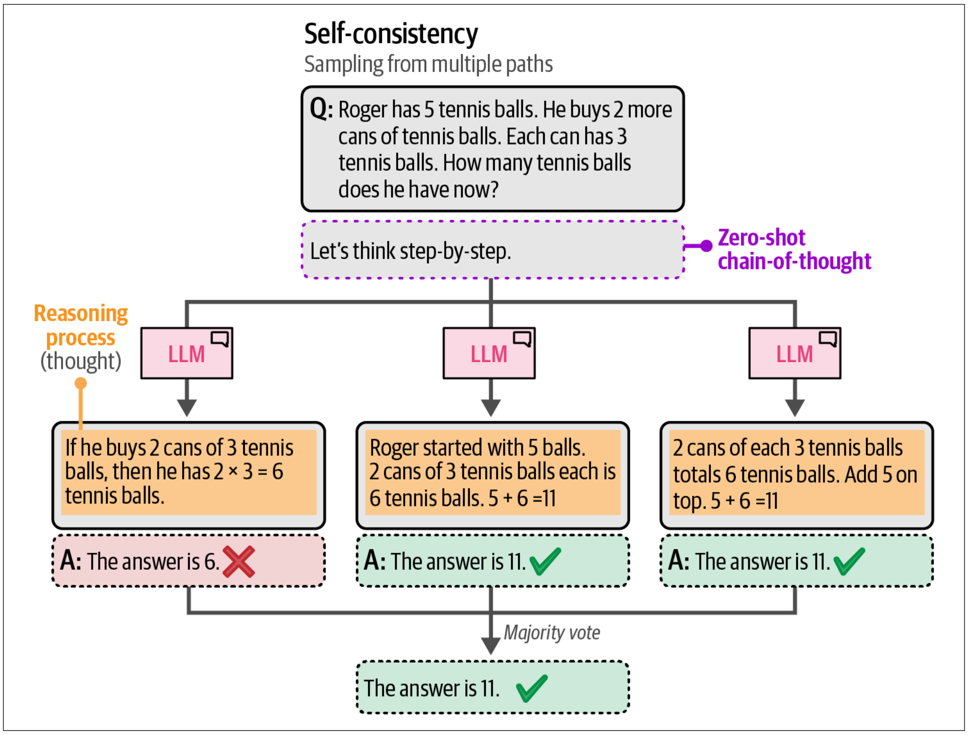

Self-consistency
Sampling from multiple paths
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
Let's think step-by-step.
Zero-shot chain-of-thought
Reasoning process (thought)
LLM
LLM
LLM
If he buys 2 cans of 3 tennis balls, then he has 2 × 3 = 6 tennis balls.
Roger started with 5 balls. 2 cans of 3 tennis balls each is 6 tennis balls. 5 + 6 =11
2 cans of each 3 tennis balls totals 6 tennis balls. Add 5 on top. 5 + 6 =11
A: The answer is 6.
A: The answer is 11.
A: The answer is 11.
Majority vote
The answer is 11.

Figure 6-17. By sampling from multiple reasoning paths, we can use majority voting to extract the most likely answer.

However, this does require a single question to be asked multiple times. As a result, although the method can improve performance, it becomes *n* times slower where *n* is the number of output samples.

Tree-of-Thought: Exploring Intermediate Steps

The ideas of chain-of-thought and self-consistency are meant to enable more complex reasoning. By sampling from multiple "thoughts" and making them more thoughtful, we aim to improve the output of generative models.

These techniques only scratch the surface of what is currently being done to mimic complex reasoning. An improvement to these approaches can be found in tree-of-thought, which allows for an in-depth exploration of several ideas.

The method works as follows. When faced with a problem that requires multiple reasoning steps, it often helps to break it down into pieces. At each step, and as illustrated in Figure 6-18, the generative model is prompted to explore different solutions to the problem at hand. It then votes for the best solution and continues to the next step.[10]

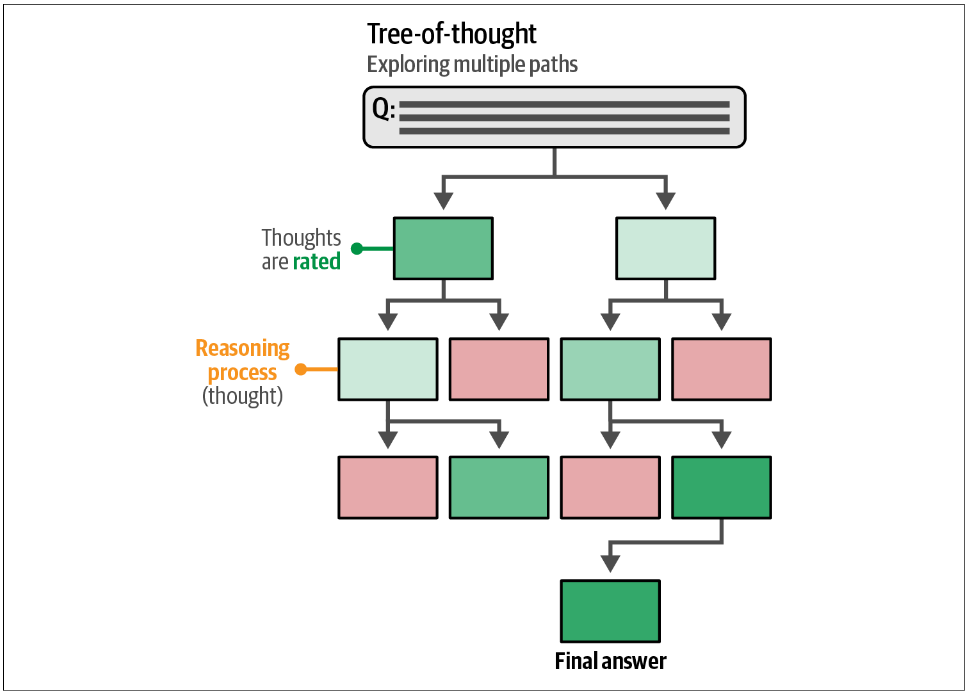


Figure 6-18. By leveraging a tree-based structure, generative models can generate intermediate thoughts to be rated. The most promising thoughts are kept and the lowest are pruned.

This method is tremendously helpful when needing to consider multiple paths, like when writing a story or coming up with creative ideas. A disadvantage of this method is that it requires many calls to the generative models, which slows the application significantly. Fortunately, there has been a successful attempt to convert the tree-of-thought framework into a simple prompting technique.[11]

Instead of calling the generative model multiple times, we ask the model to mimic that behavior by emulating a conversation between multiple experts. These experts will question each other until they reach a consensus. An example of a tree-of-thought prompt is:

10 Shunyu Yao et al. "Tree of thoughts: Deliberate problem solving with large language models." *arXiv preprint arXiv:2305.10601* (2023).

11 "Using tree-of-thought prompting to boost ChatGPT's reasoning." Available at *https://oreil.ly/a_Nos*.

```
# Zero-shot tree-of-thought
zeroshot_tot_prompt = [
    {"role": "user", "content": "Imagine three different experts are answering
this question. All experts will write down 1 step of their thinking, then share
it with the group. Then all experts will go on to the next step, etc. If any
expert realizes they're wrong at any point then they leave. The question is
'The cafeteria had 23 apples. If they used 20 to make lunch and bought 6 more,
how many apples do they have?' Make sure to discuss the results."}
]
```

We can use this prompt to explore how an LLM might respond to complex questions:

```
# Generate the output
outputs = pipe(zeroshot_tot_prompt)
print(outputs[0]["generated_text"])
```

```
Expert 1: Step 1 - Start with the initial number of apples: 23 apples.

Expert 2: Step 1 - Subtract the apples used for lunch: 23 - 20 = 3 apples
remaining.

Expert 3: Step 1 - Add the newly bought apples: 3 + 6 = 9 apples.


Expert 1: Step 2 - Confirm the final count: The cafeteria has 9 apples.

Expert 2: Step 2 - Review the calculations: 23 - 20 = 3, then 3 + 6 = 9. The
calculations are correct.

Expert 3: Step 2 - Agree with the result: The cafeteria indeed has 9 apples.


All experts agree on the final count: The cafeteria has 9 apples.
```

We again get the correct answer but instead through a "discussion between experts." It is interesting to see such a conservation between "experts" that demonstrates the creativity that comes with prompt engineering.

Output Verification

Systems and applications built with generative models might eventually end up in production. When that happens, it is important that we verify and control the output of the model to prevent breaking the application and to create a robust generative AI application.

Reasons for validating the output might include:

Structured output
: By default, most generative models create free-form text without adhering to specific structures other than those defined by natural language. Some use cases require their output to be structured in certain formats, like JSON.

Valid output
: Even if we allow the model to generate structured output, it still has the capability to freely generate its content. For instance, when a model is asked to output either one of two choices, it should not come up with a third.

Ethics
: Some open source generative models have no guardrails and will generate outputs that do not consider safety or ethical considerations. For instance, use cases might require the output to be free of profanity, personally identifiable information (PII), bias, cultural stereotypes, etc.

Accuracy
: Many use cases require the output to adhere to certain standards or performance. The aim is to double-check whether the generated information is factually accurate, coherent, or free from hallucination.

Controlling the output of a generative model, as we explored with parameters like `top_p` and `temperature`, is not an easy feat. These models require help to generate consistent output conforming to certain guidelines.

Generally, there are three ways of controlling the output of a generative model:

Examples
: Provide a number of examples of the expected output.

Grammar
: Control the token selection process.

Fine-tuning
: Tune a model on data that contains the expected output.

In this section, we will go through the first two methods. The third, fine-tuning a model, is left for Chapter 12 where we will go in depth into fine-tuning methods.

Providing Examples

A simple and straightforward method to fix the output is to provide the generative model with examples of what the output should look like. As we explored before, few-shot learning is a helpful technique that guides the output of the generative model. This method can be generalized to guide the structure of the output as well.

For example, let us consider an example where we want the generative model to create a character profile for an RPG game. We start by using no examples:

```
# Zero-shot learning: Providing no examples
zeroshot_prompt = [
    {"role": "user", "content": "Create a character profile for an RPG game in
JSON format."}
]

# Generate the output
outputs = pipe(zeroshot_prompt)
print(outputs[0]["generated_text"])
```

~~~
 ```json
{
  "characterProfile": {
    "name": "Eldrin Stormbringer",
    "class": "Warlock",
    "race": "Half-Elf",
    "age": 27,
   "gender": "Male",
    "alignment": "Chaotic Good",
    "background": "Rogue",
    …
    },
    "attributes": {
      "strength": 10,
      "dexterity": 17,
      "constitution": 12,
      "intelligence": 12,
      "wisdom": 10,
      "charisma
~~~

The preceding truncated output is not valid JSON since the model stopped generating tokens after starting the "charisma" attribute. Moreover, we might not want certain attributes. Instead, we can provide the model with a number of examples that indicate the expected format:

```
# One-shot learning: Providing an example of the output structure
one_shot_template = """Create a short character profile for an RPG game. Make
sure to only use this format:

{
  "description": "A SHORT DESCRIPTION",
  "name": "THE CHARACTER'S NAME",
  "armor": "ONE PIECE OF ARMOR",
  "weapon": "ONE OR MORE WEAPONS"
}
"""
one_shot_prompt = [
    {"role": "user", "content": one_shot_template}
]

# Generate the output
outputs = pipe(one_shot_prompt)
print(outputs[0]["generated_text"])
```

```
{
  "description": "A cunning rogue with a mysterious past, skilled in stealth
and deception.",
  "name": "Lysandra Shadowstep",
  "armor": "Leather Cloak of the Night",
  "weapon": "Dagger of Whispers, Throwing Knives"
}
```

The model perfectly followed the example we gave it, which allows for more consistent behavior. This also demonstrates the importance of leveraging few-shot learning to improve the structure of the output and not only its content.

An important note here is that it is still up to the model whether it will adhere to your suggested format or not. Some models are better than others at following instructions.

Grammar: Constrained Sampling

Few-shot learning has a big disadvantage: we cannot explicitly prevent certain output from being generated. Although we guide the model and give it instructions, it might still not follow it entirely.

Instead, packages have been rapidly developed to constrain and validate the output of generative models, like Guidance, Guardrails, and LMQL. In part, they leverage generative models to validate their own output, as illustrated in Figure 6-19. The generative models retrieve the output as new prompts and attempt to validate it based on a number of predefined guardrails.

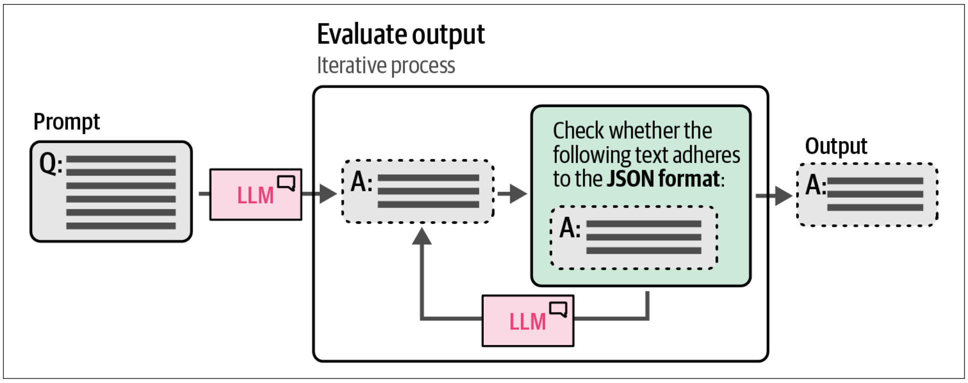

Evaluate output
Iterative process
Prompt
Q:
LLM
A:
Check whether the following text adheres to the JSON format:
A:
LLM
Output
A:

Figure 6-19. Use an LLM to check whether the output correctly follows our rules.

Similarly, as illustrated in Figure 6-20, this validation process can also be used to control the formatting of the output by generating parts of its format ourselves as we already know how it should be structured.

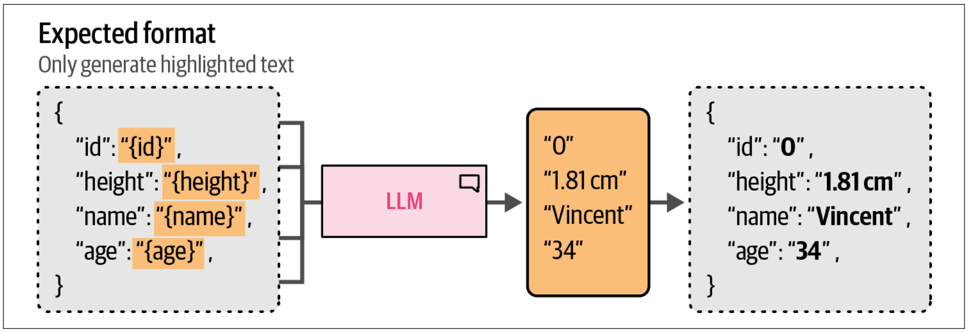


Figure 6-20. Use an LLM to generate only the pieces of information we do not know beforehand.

This process can be taken one step further and instead of validating the output we can already perform validation during the token sampling process. When sampling tokens, we can define a number of grammars or rules that the LLM should adhere to when choosing its next token. For instance, if we ask the model to either return "positive," "negative," or "neutral" when performing sentiment classification, it might still return something else. As illustrated in Figure 6-21, by constraining the sampling process, we can have the LLM only output what we are interested in. Note that this is still affected by parameters such as `top_p` and `temperature`.

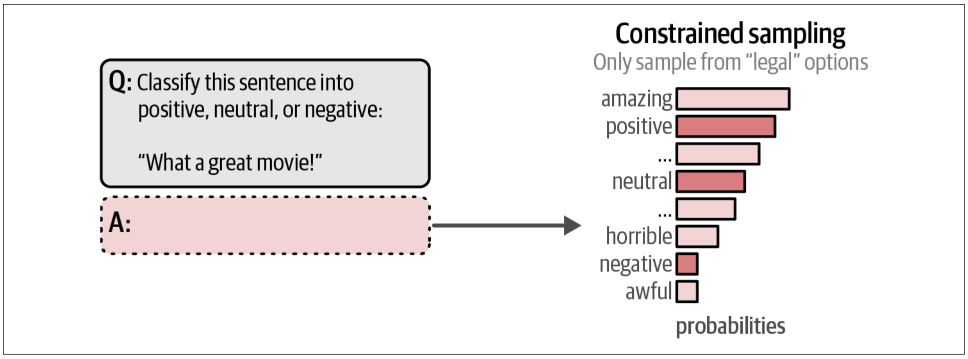


Figure 6-21. Constrain the token selection to only three possible tokens: "positive," "neutral," and "negative."

Let us illustrate this phenomenon with `llama-cpp-python`, a library similar to `transformers` that we can use to load in our language model. It is generally used to efficiently load and use compressed models (through quantization; see Chapter 12) but we can also use it to apply a JSON grammar.

We load the same model we used throughout this chapter but use a different format instead, namely GGUF. `llama-cpp-python` expects this format, which is generally used for compressed (quantized) models.

Since we are loading a new model, it is advised to restart the notebook. That will clear any previous models and empty the VRAM. You can also run the following to empty the VRAM:

```
import gc
import torch
del model, tokenizer, pipe

# Flush memory
gc.collect()
torch.cuda.empty_cache()
```

Now that we have cleared the memory, we can load Phi-3. We set `n_gpu_layers` to `-1` to indicate that we want all layers of the model to be run from the GPU. The `n_ctx` refers to the context size of the model. The `repo_id` and `filename` refer to the Hugging Face repository where the model resides:

```
from llama_cpp.llama import Llama

# Load Phi-3
llm = Llama.from_pretrained(
    repo_id="microsoft/Phi-3-mini-4k-instruct-gguf",
    filename="*fp16.gguf",
    n_gpu_layers=-1,
    n_ctx=2048,
    verbose=False
)
```

To generate the output using the internal JSON grammar, we only need to specify the `response_format` as a JSON object. Under the hood, it will apply a JSON grammar to make sure the output adheres to that format.

To illustrate, let's ask the model to create an RPG character in JSON format to be used in a Dungeons & Dragons session:

```
# Generate output
output = llm.create_chat_completion(
    messages=[
        {"role": "user", "content": "Create a warrior for an RPG in JSON for
mat."},
    ],
    response_format={"type": "json_object"},
    temperature=0,
)['choices'][0]['message']["content"]
```

To check whether the output actually is JSON, we can attempt to process it as such:

```
import json

# Format as json
json_output = json.dumps(json.loads(output), indent=4)
print(json_output)
```

```
{
    "name": "Eldrin Stormbringer",
    "class": "Warrior",
    "level": 10,
    "attributes": {
        "strength": 18,
        "dexterity": 12,
        "constitution": 16,
        "intelligence": 9,
        "wisdom": 14,
        "charisma": 10
    },
    "skills": {
        "melee_combat": {
            "weapon_mastery": 20,
            "armor_class": 18,
            "hit_points": 35
        },
        "defense": {
            "shield_skill": 17,
            "block_chance": 90
        },
        "endurance": {
            "health_regeneration": 2,
            "stamina": 30
        }
    },
    "equipment": [
        {
            "name": "Ironclad Armor",
            "type": "Armor",
            "defense_bonus": 15
        },
        {
            "name": "Steel Greatsword",
            "type": "Weapon",
            "damage": 8,
            "critical_chance": 20
        }
    ],
    "background": "Eldrin grew up in a small village on the outskirts of a
war-torn land. Witnessing the brutality and suffering caused by conflict, he
dedicated his life to becoming a formidable warrior who could protect those
unable to defend themselves."
}
```

The output is properly formatted as JSON. This allows us to more confidently use generative models in applications where we expect the output to adhere to certain formats.

Summary

In this chapter, we explored the basics of using generative models through prompt engineering and output verification. We focused on the creativity and potential complexity that comes with prompt engineering. These components of a prompt are key in generating and optimizing output appropriate for different use cases.

We further explored advanced prompt engineering techniques such as in-context learning and chain-of-thought. These methods involve guiding generative models to reason through complex problems by providing examples or phrases that encourage step-by-step thinking thereby mimicking human reasoning processes.

Overall, this chapter demonstrated that prompt engineering is a crucial aspect of working with LLMs, as it allows us to effectively communicate our needs and preferences to the model. By mastering prompt engineering techniques, we can unlock some of the potential of LLMs and generate high-quality responses that meet our requirements.

The next chapter will build upon these concepts by exploring more advanced techniques for leveraging generative models. We will go beyond prompt engineering and explore how LLMs can use external memory and tools.