Prompt Engineering

CS XXX: Introduction to Large Language Models

The basic ingredients of a prompt

- 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.
- 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.

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 the prompt does not need to be more than a few words to elicit a response from the LLM.

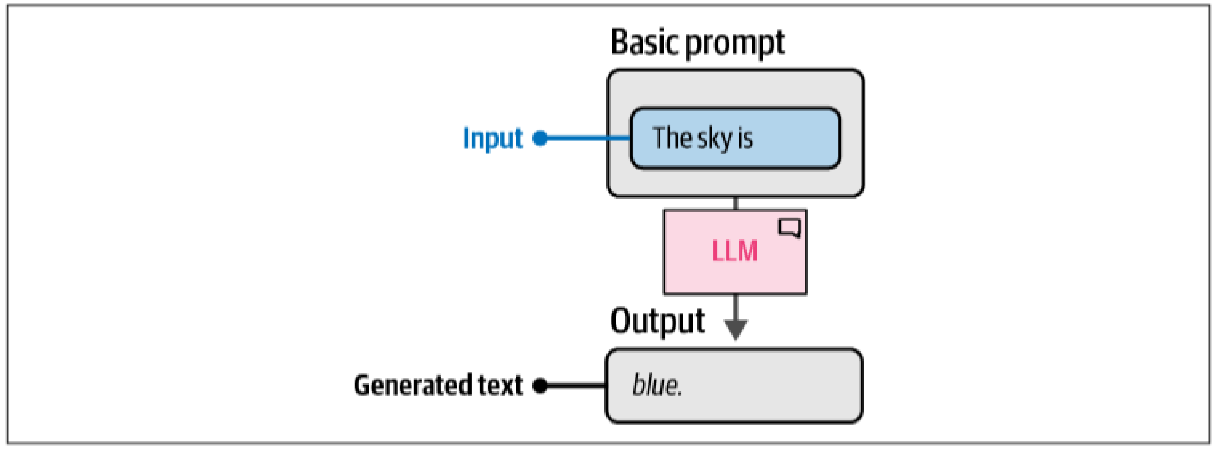


Figure 6-6. A basic example of a prompt. No instruction is given so the LLM will simply try to complete the sentence.

The basic ingredients of a prompt

- 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, 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.

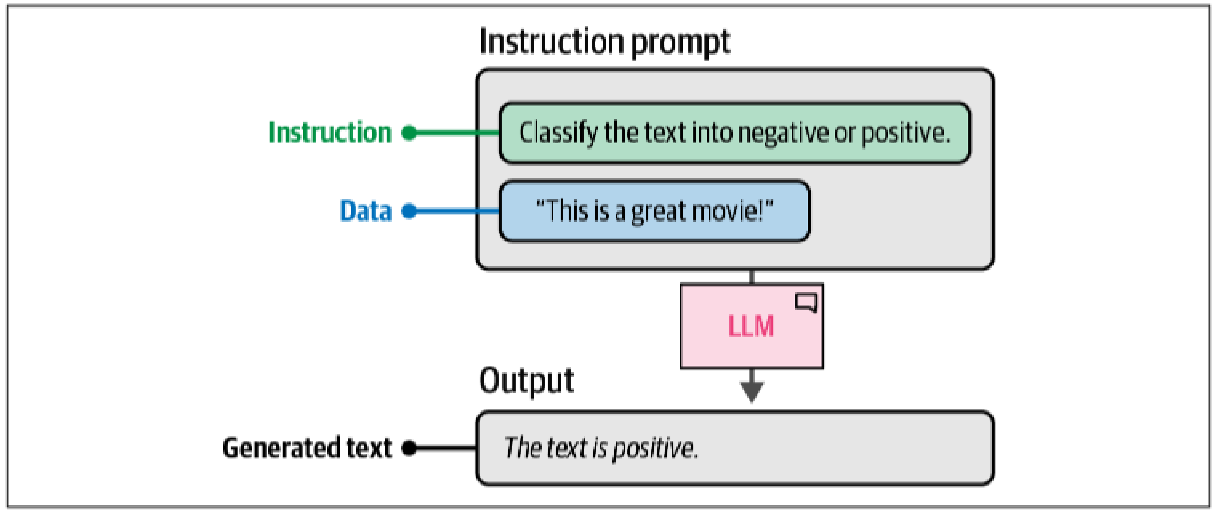


Figure 6-7. Two components of a basic instruction prompt: the instruction itself and the data it refers to.

The basic ingredients of a prompt

- 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.

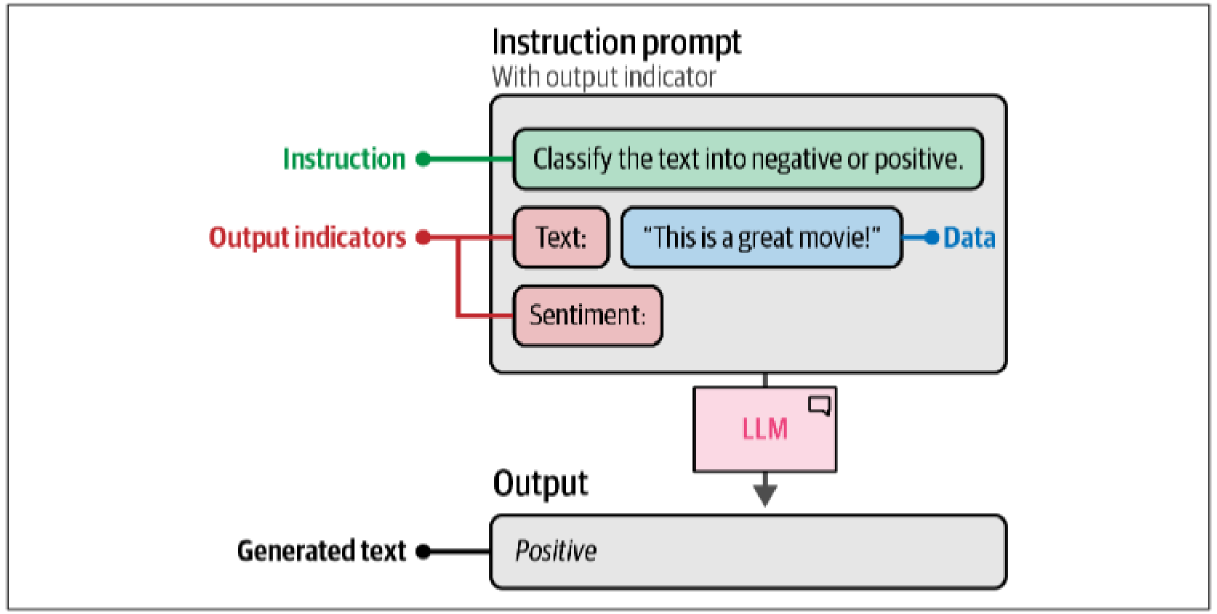


Figure 6-8. Extending the prompt with an output indicator that allows for a specific output.

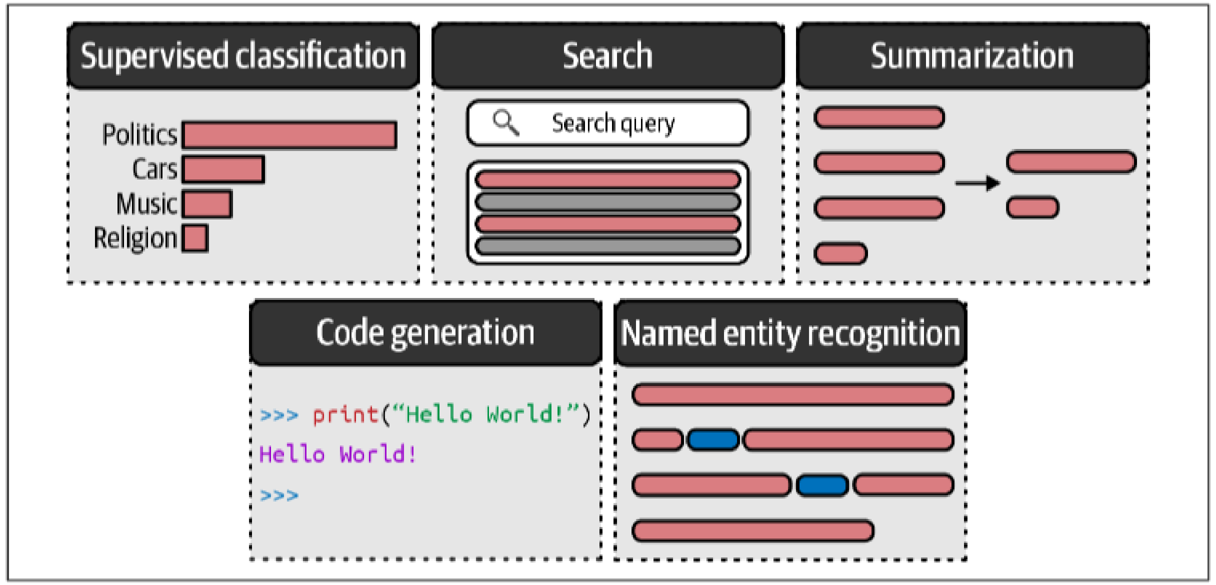
The basic ingredients of a prompt

- 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.

Instruction based prompting

- 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.

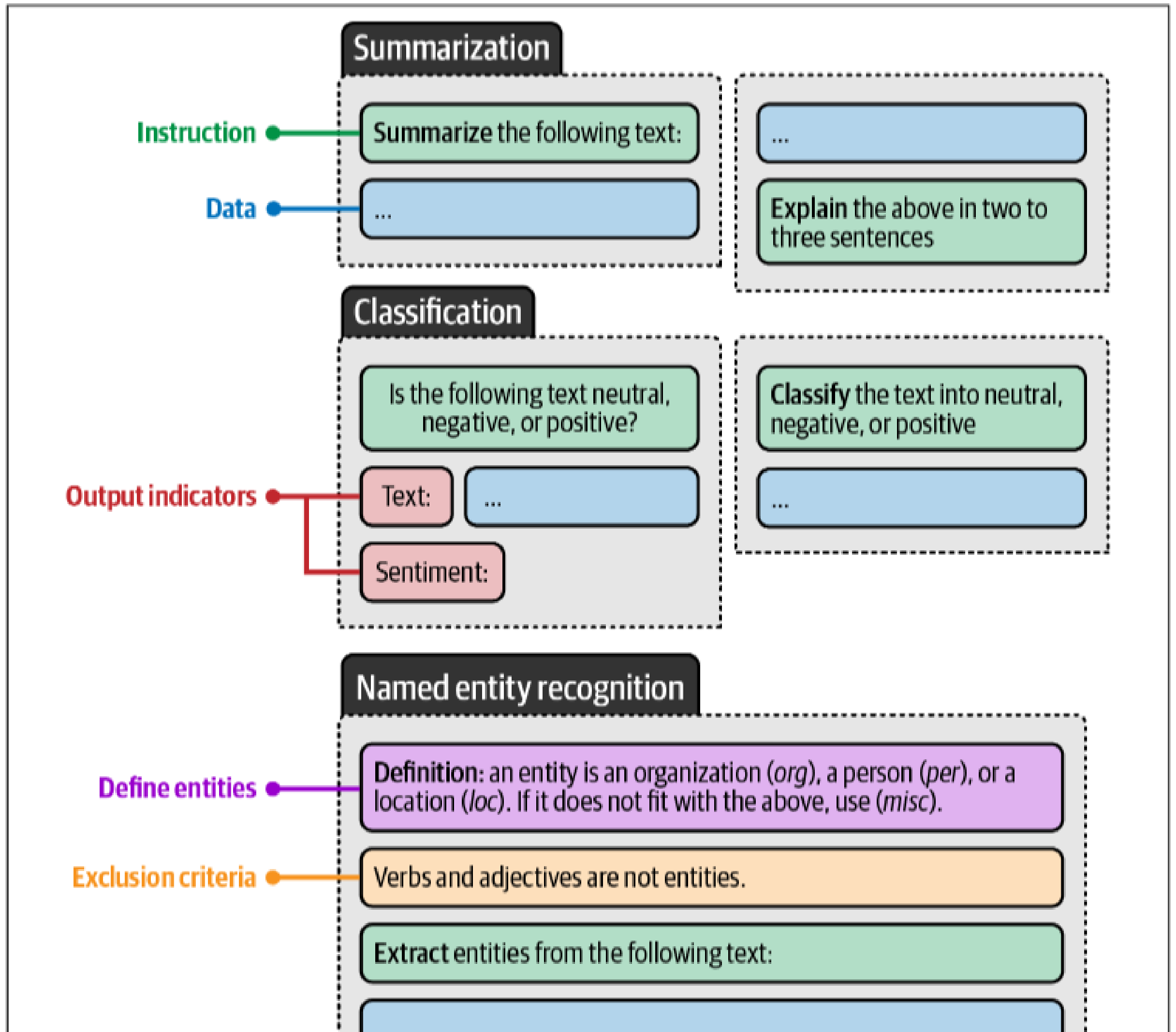


```
>>> print("Hello World!")
Hello World!
>>>
```

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

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.


Summarization
Instruction
Summarize the following text:
Data
...
...
Explain the above in two to three sentences
Classification
Is the following text neutral, negative, or positive?
Classify the text into neutral, negative, or positive
Output indicators
Text:
...
...
Sentiment:
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:

Instruction based prompting

- 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.
    - 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. LLMs tend to focus on information either at the beginning of a prompt (primacy effect) or the end of a prompt (recency effect).

- 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

- 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.

Advanced Prompt Engineering

- advanced components can quickly make a prompt quite complex
  - **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.

Advanced Prompt Engineering

- Let us extend the classification prompt we had earlier and use all of the preceding components.
- This complex prompt demonstrates the modular nature of prompting. We can add and remove components freely and judge their effect on the output.

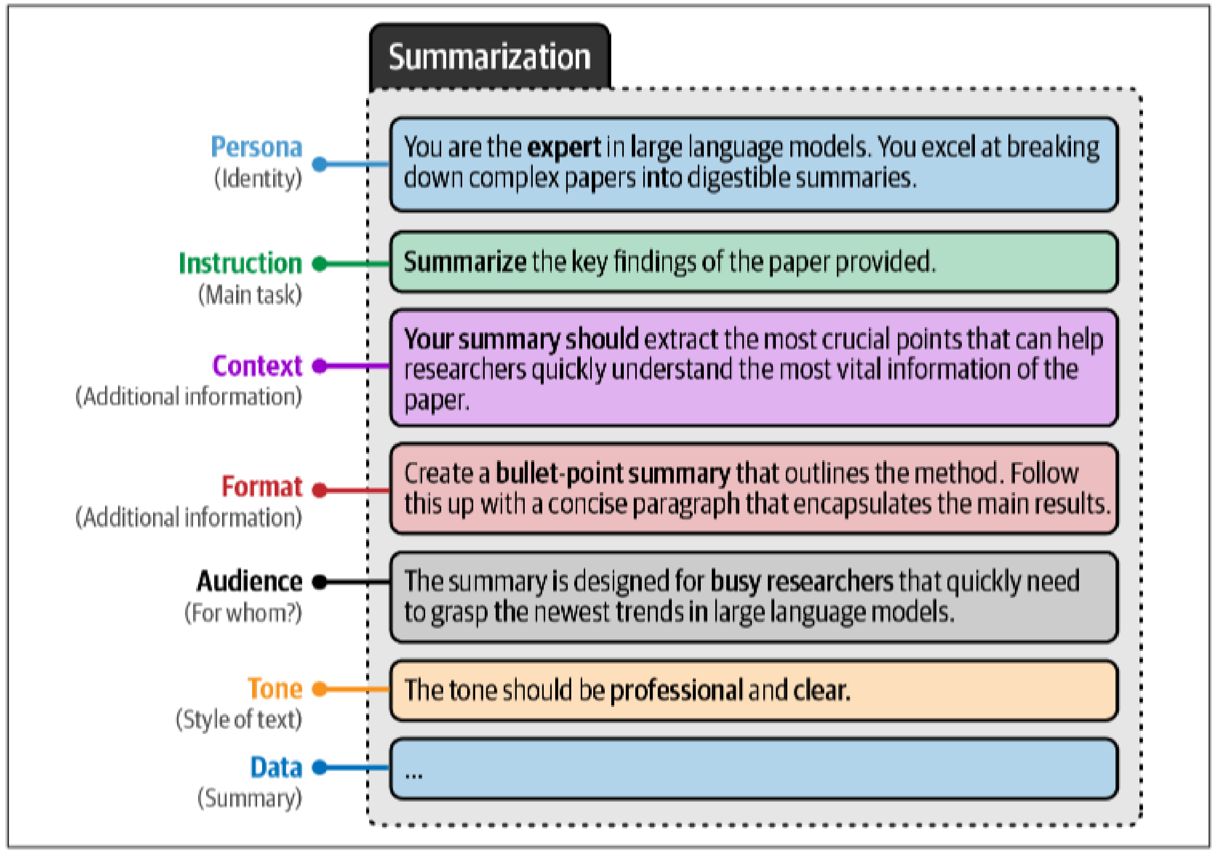

Summarization
Persona (Identity)
Instruction (Main task)
Context (Additional information)
Format (Additional information)
Audience (For whom?)
Tone (Style of text)
Data (Summary)
You are the expert in large language models. You excel at breaking down complex papers into digestible summaries.
Summarize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

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

Iterative Improvement

- 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.

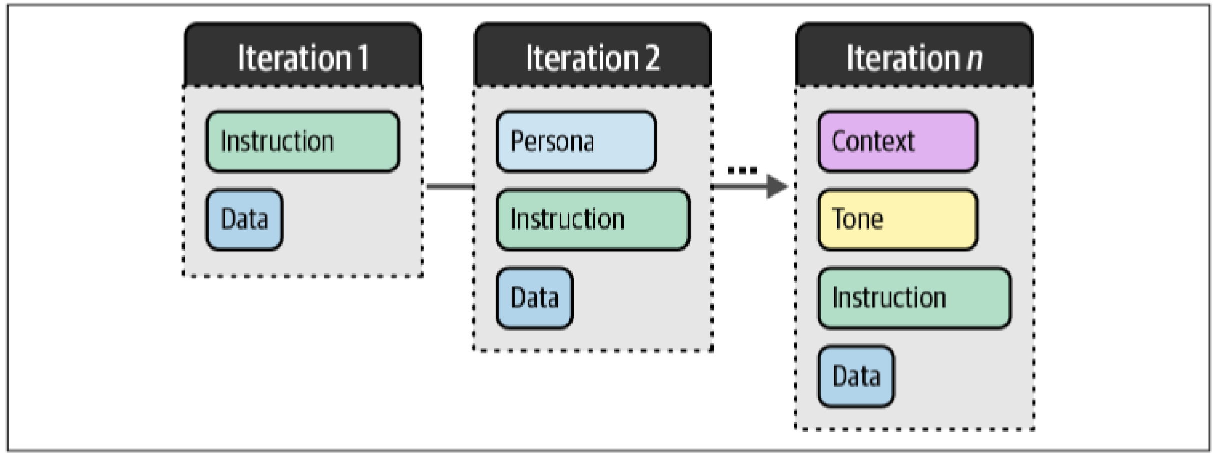


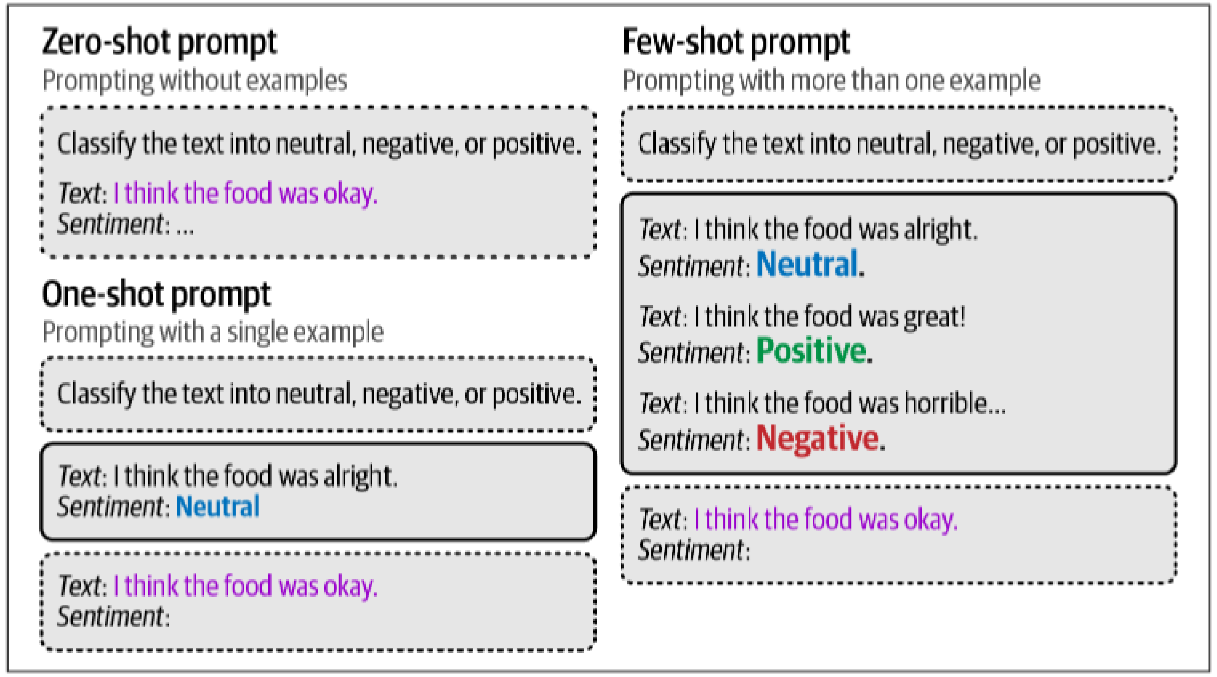
Figure 6-12. Iterating over modular components is a vital part of prompt engineering.

In-Context Learning

- 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.

In-Context Learning

- In-context learning 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.**

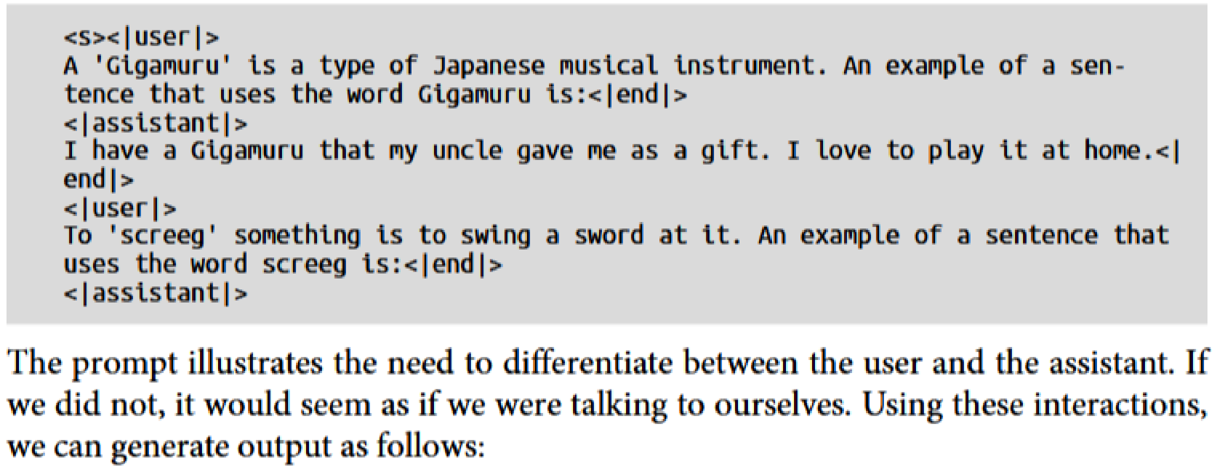
Text: I think the food was okay.
Sentiment:

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

Conversations as Examples

- We can assign roles in the prompt to provide an example response to the LLM to show exactly the kind of response we want to achieve. To do so, we will need to differentiate between our question (user) and the answers that were provided by the model (assistant).

- 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:“
- },
- {
- "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:"
- }
-]



```
<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:

Chain Prompting: Breaking up the Problem

- 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.

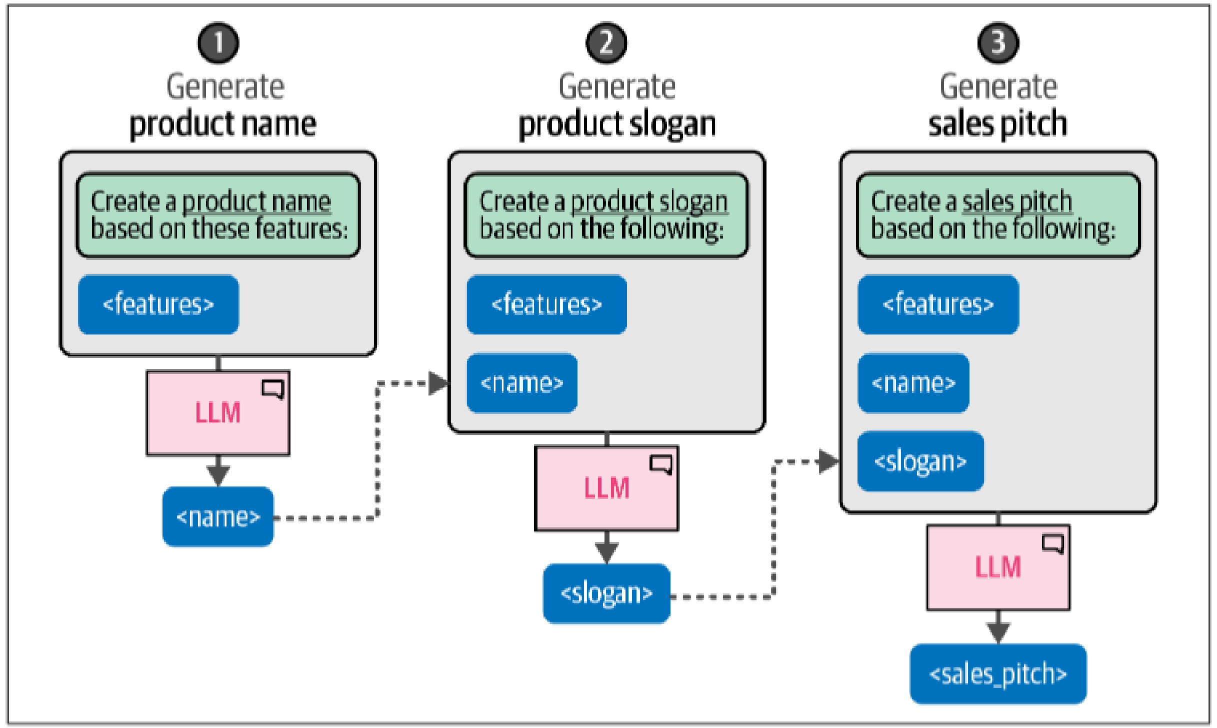
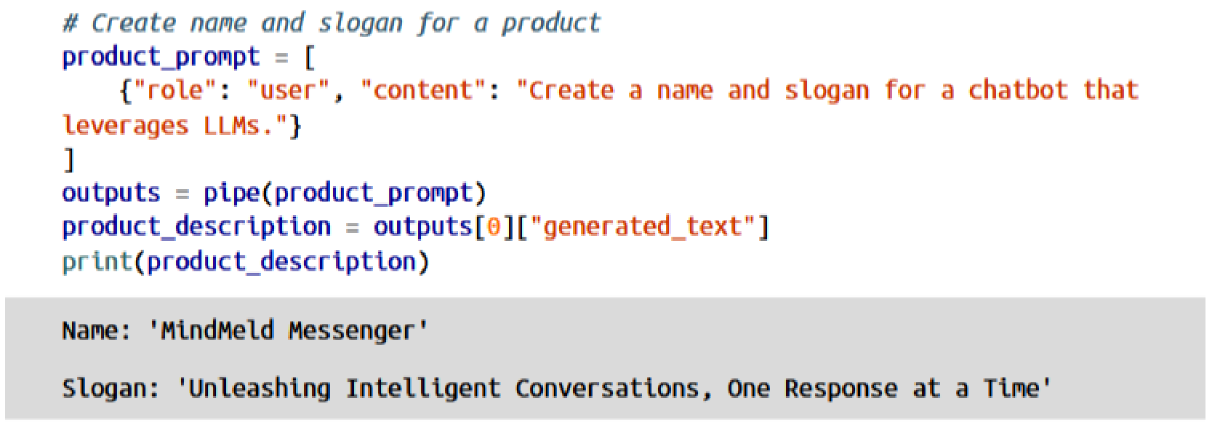


Figure 6-14. Using a description of a product's features, chain prompts to create a suitable name, slogan, and sales pitch

Chain Prompting: Breaking up the Problem

- 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:



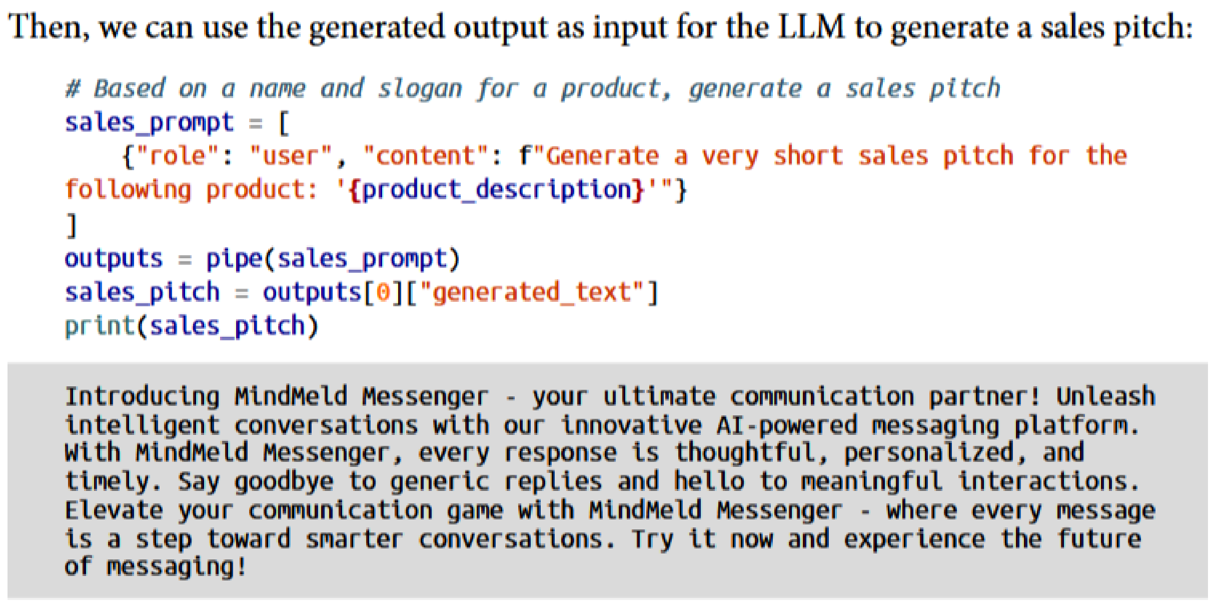
```python
# Create name and slogan for a product
product_prompt = [
    {"role": "user", "content": "Create a name and slogan for a chatbot that
leverages LLMs."}
]
outputs = pipe(product_prompt)
product_description = outputs[0]["generated_text"]
print(product_description)
```

```
Name: 'MindMeld Messenger'

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

Chain Prompting: Breaking up the Problem

- 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.



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!
```

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.

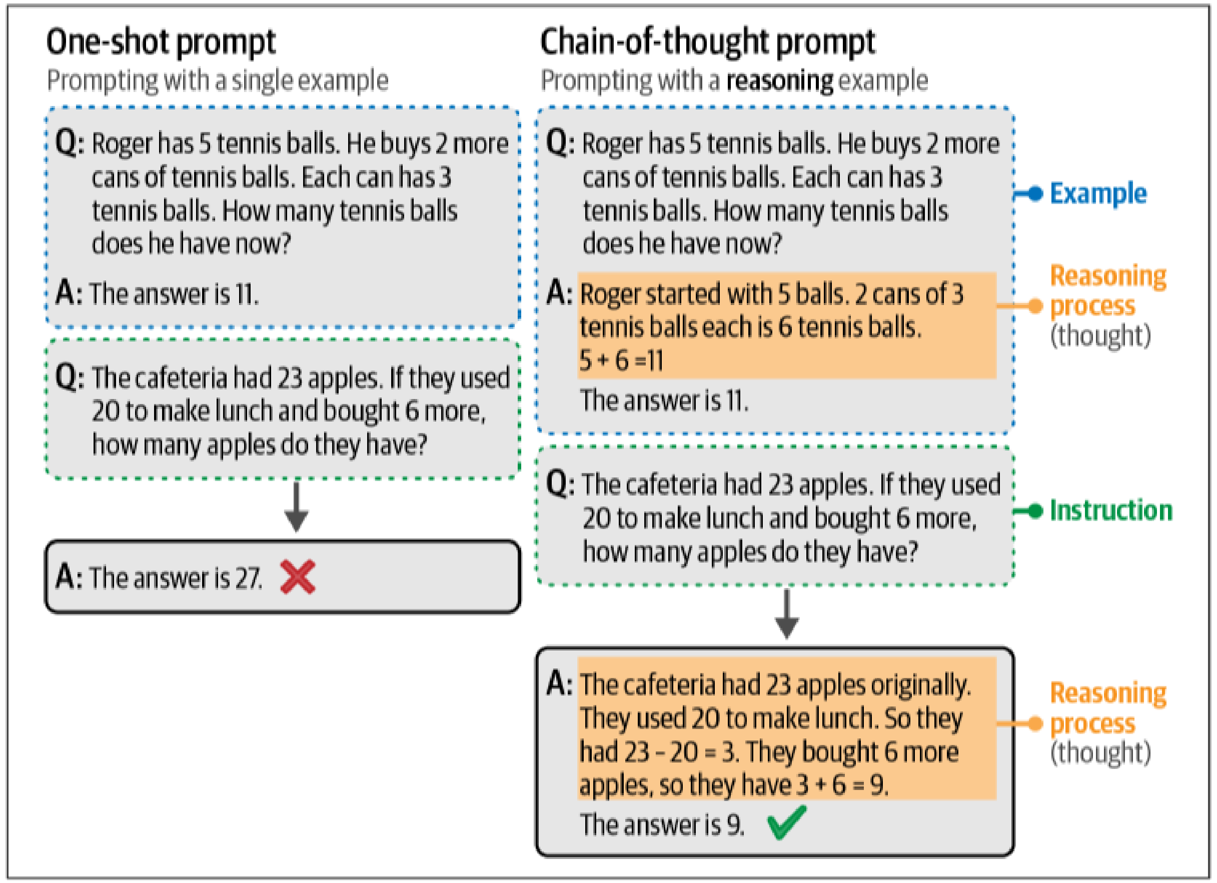
- The output that they showcase, however, can demonstrate complex behavior and although it might not be “true” reasoning, they are still referred to as reasoning capa-bilities. In other words, we work together with the LLM through prompt engineering so we can mimic reasoning processes in order to improve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

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.

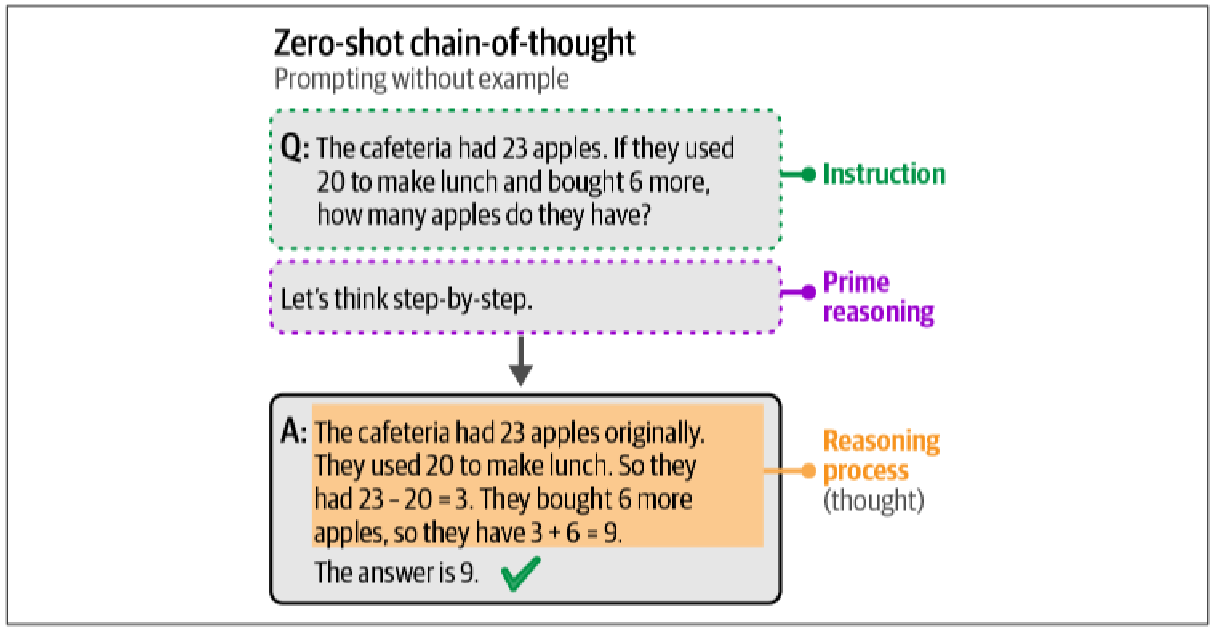
Chain-of-Thought: Think Before Answering

- 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.


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)

Chain-of-Thought: Think Before Answering

- 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"



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.

Chain-of-Thought: Think Before Answering

- Using the same prompt multiple times can lead to different results. To counteract this self-consistency was introduced. This method asks the generative model the same prompt multiple times and takes the majority result as the final answer.

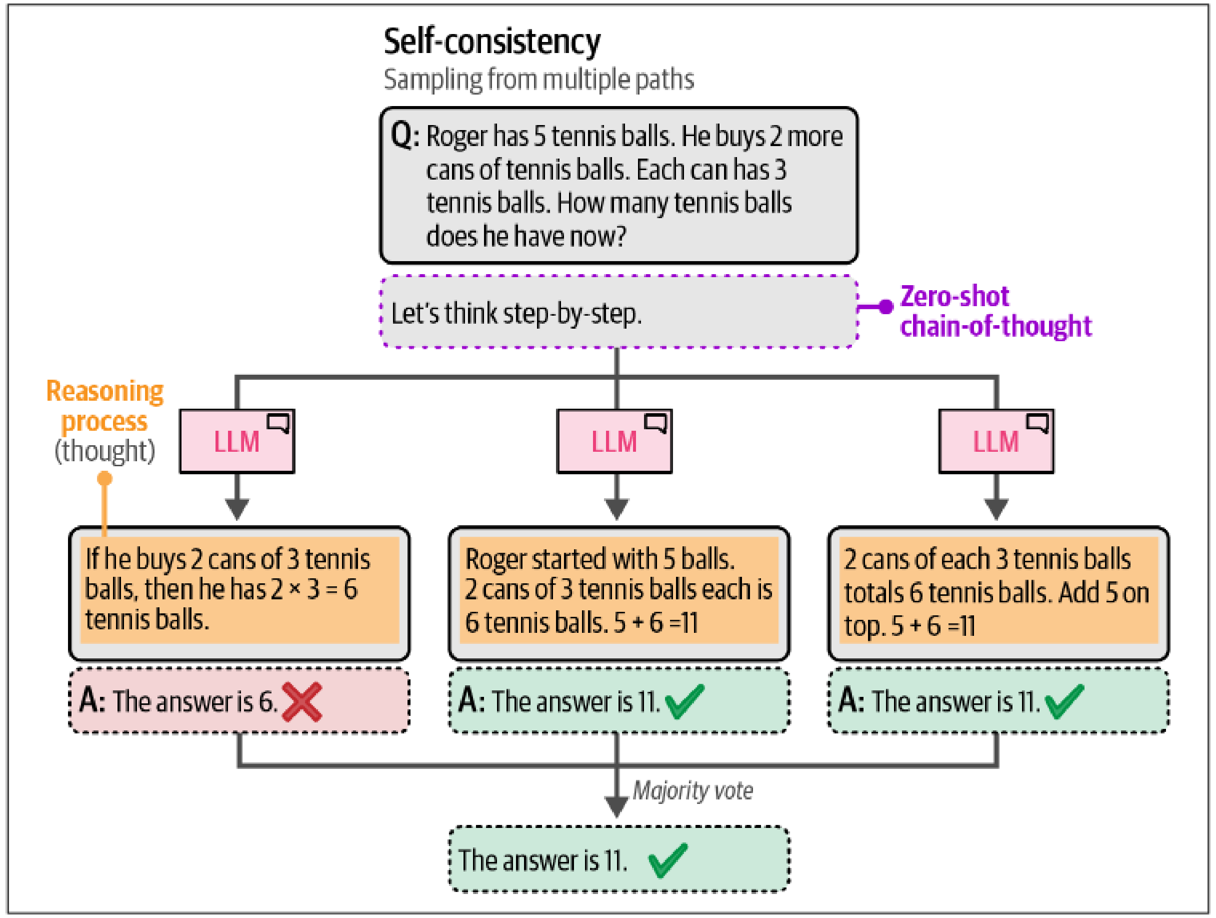

Self-consistency
Sampling from multiple paths
Q: Roger has 5 tennis balls. He buys 2 more cans of tennis balls. Each can has 3 tennis balls. How many tennis balls does he have now?
Let's think step-by-step.
Zero-shot chain-of-thought
Reasoning process (thought)
LLM
LLM
LLM
If he buys 2 cans of 3 tennis 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.

Tree-of-Thought: Exploring Intermediate Steps

- Using the same prompt multiple times can lead to different results. To counteract this self-consistency was introduced. This method asks the generative model the same
  prompt multiple tin
  he final answer. An
  improvement to thes
  ught, which allows for
  an in-depth explorati
- The method works
  that requires multiple
  reasoning steps, it
  es. At each step the
  generative model is
  ne problem at hand. It
  then votes for the be

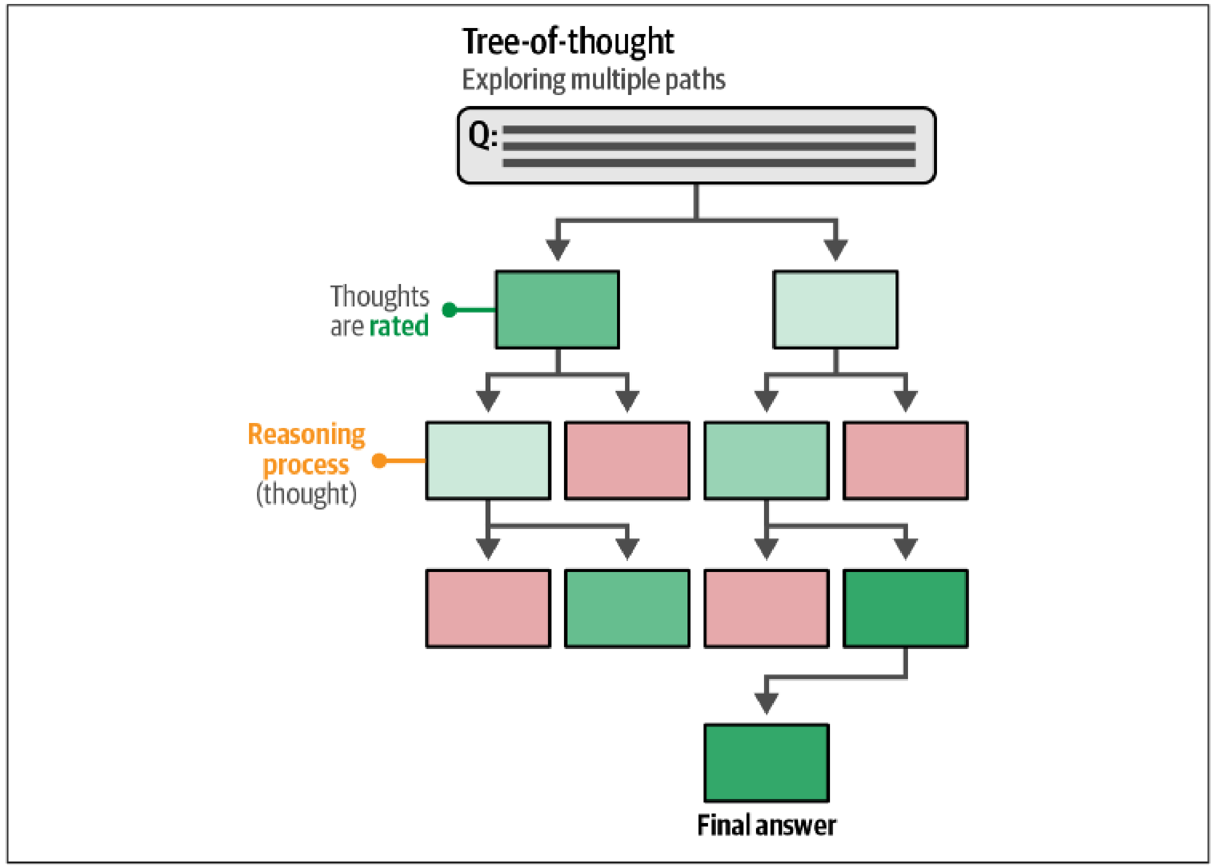
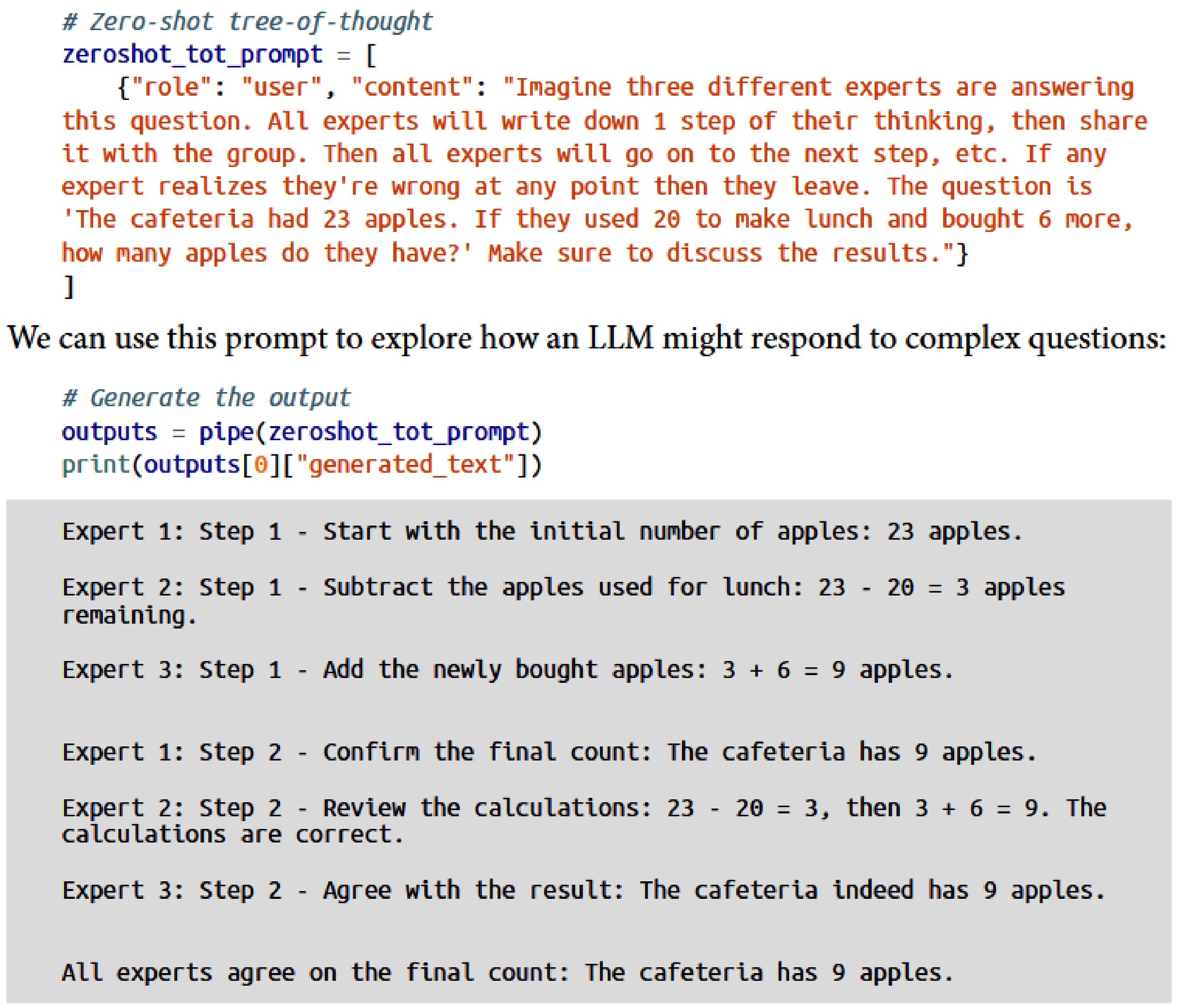


Figure 6-18. By leveraging a tree-based structure, generative models can generate intermediate thoughts to be rated. The most promising thoughts are kept and the lowest are pruned.

Tree-of-Thought: Exploring Intermediate Steps

- 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:



```
# 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.
```

Output Verification

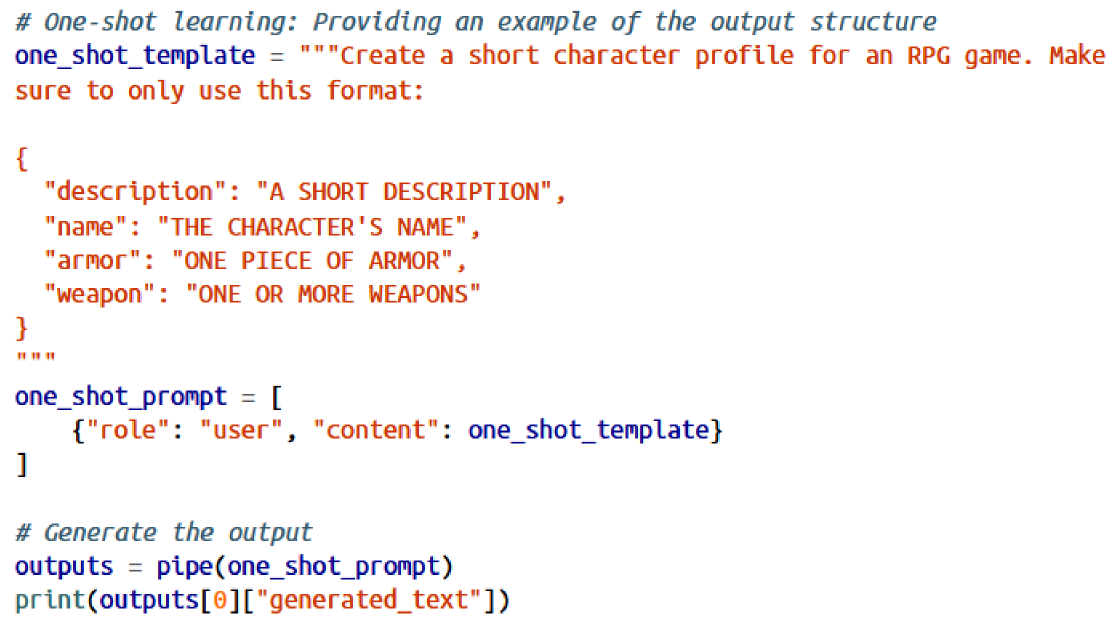
- Systems and applications built with generative models might eventually end up in production. When that happens, it is important that we verify and control the output of the model to prevent breaking the application and to create a robust generative AI application.
- Reasons for validating the output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

Output Verification

- 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.

Providing Examples

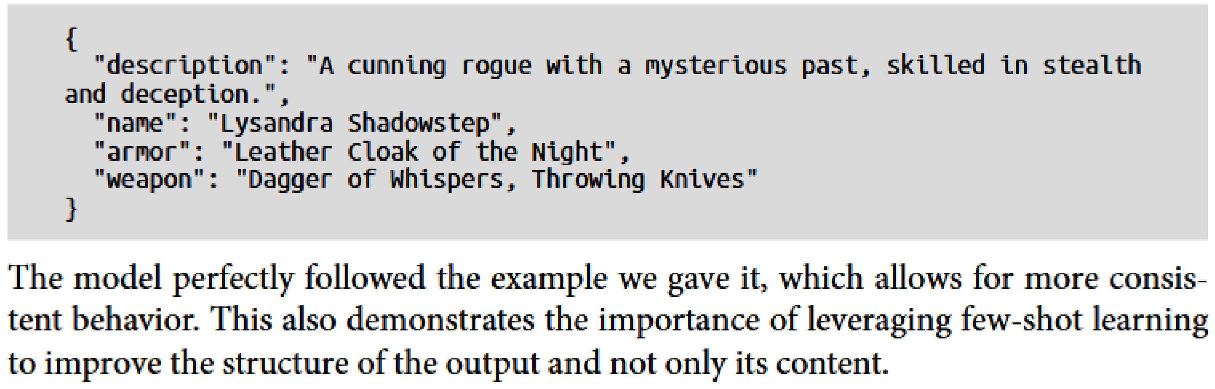
- A simple and straightforward method to fix the output is to provide the generative model with examples of what the output should look like. As we explored before, few-shot learning is a helpful technique that guides the output of the generative model. This method can be generalized to guide the structure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



```python
# 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.

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

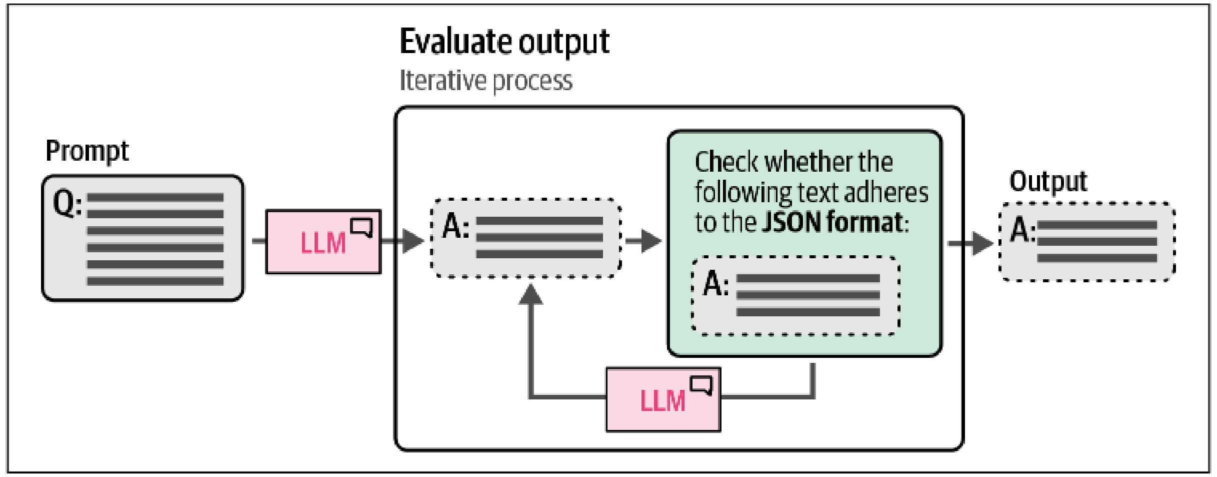

Evaluate output
Iterative process
Prompt
Q:
LLM
A:
Check whether the following text adheres to the JSON format:
A:
LLM
Output
A:

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

Grammar: Constrained Sampling

- Similarly, 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

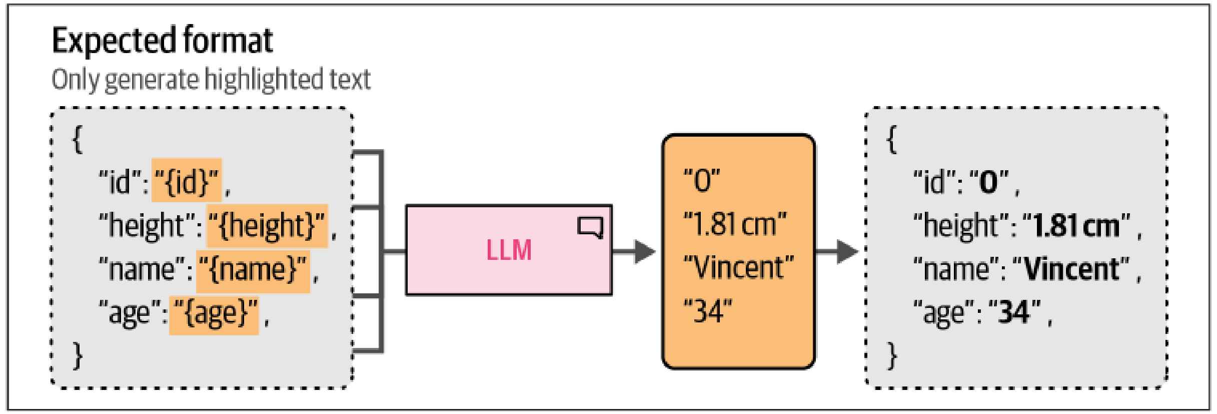


Figure 6-20. Use an LLM to generate only the pieces of information we do not know beforehand.

Grammar: Constrained Sampling

- 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. by constraining the sampling process, we can have the LLM only output what we are interested in.

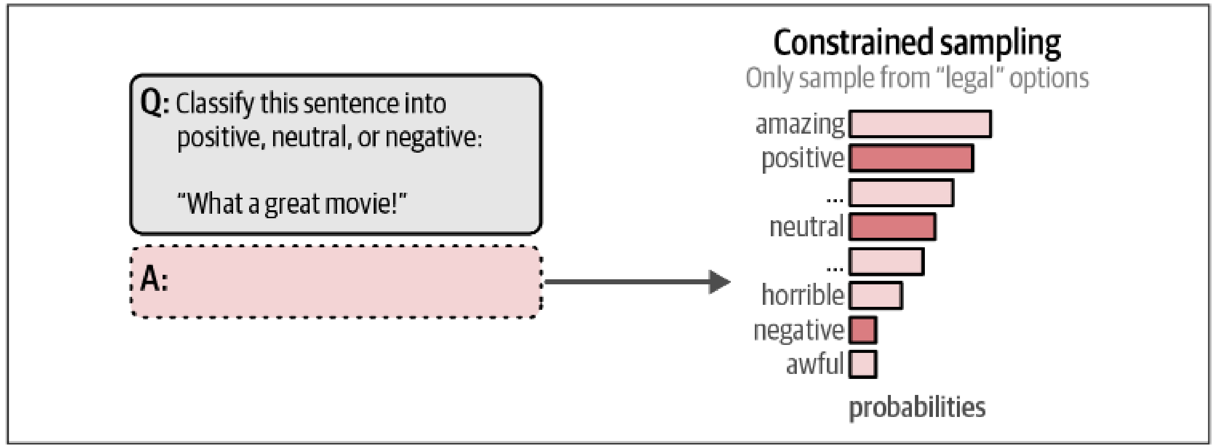


Figure 6-21. Constrain the token selection to only three possible tokens: “positive,” “neutral,” and “negative.”