

CHAPTER 6

Assembling the Prompt

In the previous chapters, you gathered a wealth of content that will serve as the building blocks for your prompt. Now, it's time to put these pieces together and craft a prompt that effectively communicates your needs. This chapter will guide you through the process of shaping your prompt by first exploring the different structures and options available to you. How you choose to organize these individual snippets will play a crucial role in the effectiveness of your final prompt.

The next step involves triaging your content—deciding what to keep and what to discard so that it will fit within any size constraints you might have. This process is key to refining your prompt and ensuring it remains focused and relevant. With your content finalized, you'll then move on to assembling your prompt, which will be your tool for eliciting relevant, coherent, and contextually accurate responses from the model. Let's dive in.

Anatomy of the Ideal Prompt

Before we go into the details of how to get there, let's visualize where we want to go. Take a look at [Figure 6-1](#), which gives a bird's eye view of how your prompt *should* look. We'll go through its elements one at a time. Concise and crisp prompts are generally more effective—plus, they use less computational power and are processed more quickly. Additionally, you have a hard cut-off with the context window size.

As discussed in [Chapter 5](#), a prompt consists of elements drawn from dynamic context and static instructions that clarify your question. There are no hard rules for the size or number of these elements. In fact, as applications evolve, a large prompt element might be broken into several smaller ones for more precise constructions. We've worked on projects with prompts ranging from just three lengthy elements to hundreds of one-line elements.

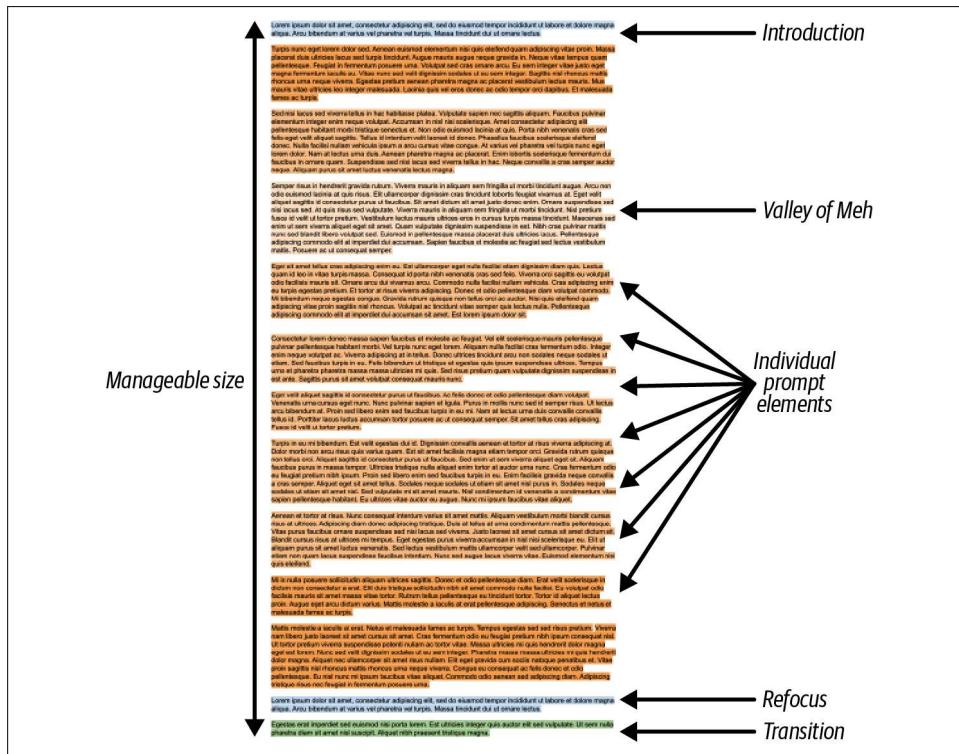


Figure 6-1. Anatomy of a well-constructed prompt

There's no theoretical rule that each prompt element must end with a newline character. However, in practice, enforcing the rule that all elements end with newlines can simplify your string manipulation code. It can also assist with token length computation, depending on the tokenizer used (more on that next). If your prompt elements don't easily fit this format, don't feel compelled to force it.

Most prompts include a handful of certain elements. First is the *introduction*, which helps you clarify the type of document you're writing and sets up the model to approach the rest of the content correctly. The introduction sets the context for everything that follows. For example, if the model states, “This is about recommending a book,” it will focus on relevant aspects for book recommendations and interpret the context accordingly. The introduction also lets the model begin thinking about the problem from the start. Since the model has a fixed “thought budget” per token and can't pause for deeper reflection, guiding its focus early on can improve its output.

Most prompts have only one introduction to set up the main question. But the principle also applies to subsections of the prompt: if there are some pieces of context where the model needs to focus on a certain aspect, it helps if you set up that aspect in the beginning.

After the introduction, you'll see a long parade of different prompt elements. The model will try to make good use of all of them, but not equally. All LLMs are subject to two effects:

In-context learning

The closer a piece of information is to the end of the prompt, the more impact it has on the model.

The lost middle phenomenon

While the model can easily recall the beginning and end of the prompt, it struggles with the information stuffed in the middle.

These two dynamics create the *Valley of Meh*, as we like to call it. The valley lies around the early middle of the prompt, and the context that is placed there is not used as effectively as the context in the beginning or the second half of the document. How deep the Valley of Meh is and its exact location depends on the model, but all models have it—as do humans!

The Valley of Meh is most problematic with large prompts, and there's no perfect solution. You can reduce its impact by placing key, high-quality prompt elements outside the Valley of Meh and by filtering context to keep the prompt as concise as possible.

When you've included all the context, it's time to remind the model of the main question. We call this the *refocus*, which is necessary for longer prompts, where you've spent a long time adding context and you need to focus the model's attention back to the question. Most prompt engineers use the *sandwich technique*, in which they start and end the prompt by clearly stating what they want the model to do (see [Table 6-1](#)).

Table 6-1. Sandwiching the context between two versions of the same question for a model with the ChatML API

Prompt part	Sandwich	Prompt
Introduction		[{"role": "system", "content" : "You are a helpful AI."}, {"role": "user", "content" : "I want to suggest to Fiona an idea for her next book to read."}]
	Sandwich part 1	Please ask any questions you need to arrive at an informed suggestion.}, {"role": "assistant", "content" : "Of course! The following information might be useful: What books did she read last?"},

Prompt part	Sandwich	Prompt
Context		<pre>{"role": "user", "content" : "Harry Potter, Lioness Rampant, Mr Lemoncello's Library"},</pre>
		<pre>{"role": "assistant", "content" : "What did she post on social media recently?"}, {"role": "user", "content" : [...]}</pre>
		[...]
		[...]
		[...]
Refocus + Transition		<pre>{"role": "assistant", "content" : "I believe this is all the information I need to select a single best candidate book suggestion."},</pre>
	Sandwich part 2	<pre>{"role": "user", "content" : "Excellent! So based on this, which book should I suggest to her?"}]</pre>

The refocus can be as short as half a line, but it's common to include key clarifications here. The introduction sets the stage ("I'm thinking about book suggestions for X."), while the refocus gives clear details ("What's the best book to recommend next, focusing on narrative prose currently available?"). If the clarification gets lengthy, you might need a brief refocus at the end, especially when discussing the output format.

The very last part of your prompt should firmly transition from explaining the problem to solving the problem—that's the bit you want the LLM to help with, after all. It's not helpful if it just keeps adding more (likely made-up) context to your main question.

When using a chatlike interface, this part is usually as simple as including a question mark at the end. RLHF has drilled those models to respond by solving the last question stated—or sometimes even just implied—in their input. Some commercial platforms, like OpenAI's ChatGPT, automatically signal when the assistant should begin its response after receiving a prompt through their API. However, traditional completion models require more explicit guidance to achieve the same effect.

The most common way to transition—especially when using a completion API—is to change your perspective from problem poser to problem solver and begin writing the answer for the model. This way, the model has no choice but to present its solution. [Figure 6-2](#) demonstrates the difference a good transition can make in getting an answer out of the model. Note that the opening quotation mark ending the transition in column three is still part of the prompt.

I'm wondering which book to suggest to Fiona.	I'm wondering which book to suggest to Fiona.	I'm wondering which book to suggest to Fiona.
I'm aware that she likes cats.	I'm aware that she likes cats.	I'm aware that she likes cats.
I'm aware that she liked Harry Potter.	I'm aware that she liked Harry Potter.	I'm aware that she liked Harry Potter.
I'm aware that she's 26 years old.	I'm aware that she's 26 years old.	I'm aware that she's 26 years old.
I'm aware that her favorite color is blue.	I'm aware that her favorite color is blue.	I'm aware that her favorite color is blue.
I'm aware that she likes to read.	What should I suggest as her next book? I'm thinking of suggesting a book that she can read in one sitting.	Based on these, I believe the book she should read next is "The Cat Who Came for Christmas" by Cleveland Amory.

Figure 6-2. Three variations of transition: missing, on the left; naive, in the middle; and refined, on the right (all completions [shaded backgrounds] obtained using OpenAI's text-davinci-002, which is a completion model, rather than a chat model)

As shown in Figure 6-2, you can often merge the refocus and the transition. In those cases, you write the beginning of the answer that is just restating or summarizing the problem statement. The actual answer is then supplied by the model.

What Kind of Document?

A prompt and completion together form a document, and as the Little Red Riding Hood principle from Chapter 4 suggests, it's best to use documents similar to those in the training data so that the format of the completion will be easy to anticipate. But what type of document should you aim for? There are several useful types, each with room for personalization. Let's explore the most common ones and when to use each.

The Advice Conversation

In the most common archetype, your document represents a conversation between two people. One asks for some kind of help, and the other provides it. The one who asks for help represents either your application or its user, while the model will take on the role of help provider.

This approach is ideal for chat models, but even completion models can benefit from it. In fact, OpenAI developed ChatML to focus on advice conversations because they believed they were the most universally useful and easiest to implement. Advice conversations have many advantages, including the following:

Natural interaction

It's easy for people to think in terms of conversations. You can pose a question directly to the model and take its continuation as the answer to simplify interactions.

Multiround interactions

For complex interactions, you can continue the prompt with new questions and answers, making it easier to manage and break down the conversation. This approach allows you to add your logic between questions and helps the model handle each query directly.

Real-world integration

Conversations work well for multiround processes and integrating with real-world tools and techniques, whether you're using a chat model or a completion model with conversational documents.

If you use this structure with a chat model, you'll get the additional advantages of RLHF related to compliance with your instructions. But if you use it with a completion model instead, you can avoid any RLHF trait that's unhelpful for your scenario (e.g., stylistic habits, content policing).

On the other hand, if you happen to be using a completion model, you can use a trick called *inception*, in which you dictate the beginning of the answer. Do you remember the 2010 movie *Inception*? Same idea here—start the answer for the model and the model will think that it's the one that came up with it, and it will generate the rest of the completion accordingly. This approach can improve model compliance and make answers easier to parse, as well as help you avoid uncertainty about whether the response will start with a general statement or get straight to the point.

When writing a prompt for a completion model, you'll have to decide the format for your transcript. The good news is that LLMs are used to many different formats, as shown in [Table 6-2](#), in which the same conversation is cast into different formats. Note that, typically, an application would provide more context for this question.

Table 6-2. Asking an LLM for ideas to plan your day

Format	Freeform text	Script format	Markerless	Structured
Introduction	I asked my husband, "What shall we do tomorrow?"	Me: What shall we do tomorrow?	What shall we do tomorrow?	<me> What shall we do tomorrow? </me>

Format	Freeform text	Script format	Markerless	Structured
Context	and he replied, "Well, what's the weather like?" I reported that "We expect a balmy 75 degrees with sunshine in the whole Boston area."	Husband: Well, what's the weather like? Me: We expect a balmy 75 degrees with sunshine in the whole Boston area.	Well, what's the weather like? We expect a balmy 75 degrees with sunshine in the whole Boston area.	<husband> Well, what's the weather like? </husband> <me> We expect a balmy 75 degrees with sunshine in the whole Boston area. </me>
Refocus	He reflected a bit about what would be a good Sunday activity for us,	Husband (after reflection about good Sunday activities)	(reflects about good Sunday activities)	<direction> Husband reflects about good Sunday activities </direction>
Transition	and he suggested,	:	I suggest we should	<husband>
Completion (text-davinci-003)	"Let's go on an outing to the beach, then. We could grab some coffee and sandwiches and have a picnic on the sand. We could sit on a blanket and watch the waves. Sounds nice."	I think it would be fun to take a bike ride, maybe in one of the nearby parks. Maybe we can bring a picnic lunch and make a day of it outside in the sunshine.	take a picnic and a book down to the Charles River and spend the day lounging, reading, and enjoying the sunshine.	If it's going to be so nice, how about a hike in the Blue Hills? We can check out the reservoir, have a picnic, and watch the sunset. </husband>

While all formats are effective, each of them has unique strengths, and we've arranged them so that each format addresses the weaknesses of the one before it:

Freeform text

This allows you to insert various types of information between quotes, but it is challenging to assemble on the fly. It can be difficult to create a reliable system for dynamically generating prompts with many elements.

Transcript format

This is easy to assemble but less effective for long or formatted elements (like source code with important indentation).

Markerless format

This works well with formatted text and longer pieces (such as pasted emails), but it can be difficult for the model to track speakers and for the application to determine when the model's response ends and the next input begins.

Structured format

This clearly indicates who is speaking and when they finish. Various structures are available, and they are detailed in "[The Structured Document](#)" on page 133.

In [Chapter 3](#), we introduced the notion that writing a conversational prompt is like playwriting. Except for stage directions, all parts of the text belong to one of the "roles" in the play. In conversations between an advice seeker and an assistant, you

typically let the user write the speaking parts for the advice seeker and let the LLM write the speaking parts for the assistant. This needn’t be the case—there’s nothing stopping you, as a prompt engineer, from writing for the role of the assistant. This is another form of the inception approach—you speak for the assistant, and in all subsequent turns of the conversation, the assistant will act as if it actually said what you said.



Writing a prompt from the assistant’s perspective helps frame context as if it’s responding to a question they asked. This approach ensures that the completion starts with the answer rather than another clarifying question.

The Analytic Report

Each year, millions of students are trained in report writing. They learn the art of crafting introductions, expositions, analyses, and conclusions, and once they graduate and enter the workforce, they churn out reports that analyze markets, weigh costs and benefits, and propose actionable conclusions. All of this is hard work, and fortunately, it serves a purpose: it provides excellent training material for LLMs. And these models are trained on vast datasets filled with reports of every kind and size.

Leveraging this abundance of reports is straightforward, especially if your task falls within domains where analytical reports are common, such as business, literature, science, or law (though it’s probably best to leave legal defense to human professionals). Reports are easy to structure because they follow a familiar format that usually starts with an introduction, leads to a conclusion, and often includes a recap. The information you’ve already gathered can be easily inserted into the discussion or background sections.

However, crafting static prompt elements like instructions requires some thought, especially if you’re making sure to keep things clear and to the point. A helpful strategy is to include a *Scope* section that clearly defines the boundaries of the report. Instead of going back and forth in a dialogue to clarify exclusions (e.g., “Please suggest only novels, not self-help books.”), you can state up front, “This report focuses solely on novels, excluding self-help books.” LLMs tend to respect such clear boundaries more consistently in reports than in dialogues.

Reports also favor objective analysis, which lightens the cognitive load for the LLM by avoiding the need for it to simulate social interaction. That said, because analysis typically precedes the conclusion, you must ensure a clear transition when you want the model to shift into a decision-making mode. Otherwise, you might end up with a meandering response that requires extra parsing. On the flip side, this format lends itself well to chain-of-thought prompting, which is covered in more detail in [Chapter 8](#).

Dialogues can take many forms, depending on the context (see [Table 6-2](#)). However, for reports, we recommend that you consistently stick to one format: writing your prompts in Markdown. Here's why:

- It's pretty universal, and the internet is full of Markdown files, so LLMs know it well.
- Markdown is a simple, lightweight language with only a few key features. This makes it easy to write and straightforward for models to interpret the output.
- Markdown's headings help define hierarchy, which allows you to organize prompt elements into clear sections that can be easily rearranged or omitted while maintaining structure.
- Another useful feature is that indentation doesn't usually matter, but for technical content (e.g., source code), you can use blocks opened and closed with triple backticks (```).
- If you want to display model output to the user directly, Markdown is very easy to render.
- Markdown's hyperlink feature allows the model to include links that are easy to parse, which can help you verify sources and retrieve content programmatically.

Additionally, it's common to give a table of contents at the beginning of the Markdown file, and that can be quite useful. A table of contents can serve as a useful part of the introduction for a long prompt because it helps models orient themselves just as much as it helps people. It can also be a great tool for controlling the completion, in two ways:

1. For chain-of-thought prompting or managing overly verbose models, you can use a scratchpad approach. Adding sections like `# Ideas` or `# Analysis` before `# Conclusion` in the table of contents helps guide the model to a more informed conclusion while allowing you to ignore the earlier sections.
2. You can easily signal when the model's response should end by adding a section like `# Appendix` or `# Further Reading` after the conclusion. Setting `# Further Reading` as a stop sequence ensures the model finishes its task, conserving compute resources.

Both use cases for the table of contents are demonstrated in [Figure 6-3](#). Note that since this is an example, the amount of context is less than what the model would need to give a proper answer to such a question. Also, LLMs aren't oracles: with the appropriate context, a model can be a good tool for ideation, but there's no reason its opinion should count for more than Jerry's in Accounting.

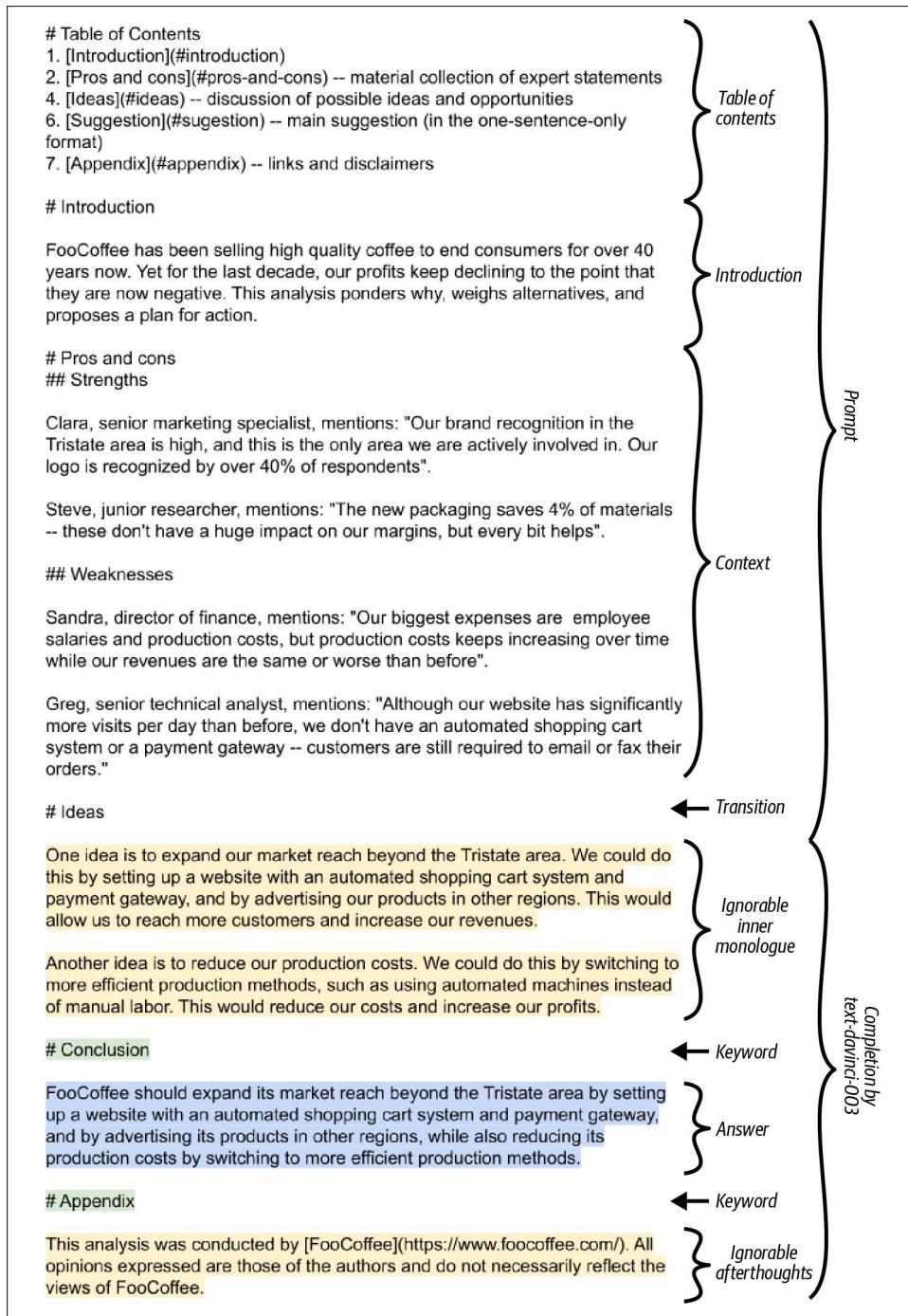


Figure 6-3. A Markdown report using a table of contents (completion obtained using OpenAI's text-davinci-003)

The Structured Document

Structured documents follow a formal specification that allows you to make strong assumptions about the form of the completion. This makes parsing easier, including the parsing of complex outputs.

A great example of this is found in Anthropic's Artifacts prompt. Artifacts will come up again in the final chapter of this book, but for now, you should know that Artifacts are self-contained documents that the user and assistant collaborate on. Examples of Artifacts are Python scripts, small React apps, mermaid diagrams, and scalable vector graphics (SVG) diagrams. Artifacts are presented in the UI as text in a pane to the right of the conversation, and in the case of React, Mermaid, and SVG, they are rendered into functioning or visual prototypes.

An abridged version of the Artifacts prompt is shown in [Table 6-3 \(this prompt was extracted by @elder_plinius\)](#). To make Artifacts work, the prompt uses an XML document structure that clearly delineates the pieces of the interaction. The `artifacts_info` prompt holds the equivalent of the system message, explaining how artifacts work. It includes an `examples` section with several `example` blocks. Each example has a `user_query` and an `assistant_response`.

Things get most interesting inside the `assistant_response`. First, the assistant starts its response, and then an `antThinking` block is injected so that the assistant can “think” about whether the user’s request should make use of an Artifact or just be handled as a normal conversation. If the decision is to use an Artifact, then the text will include an `antArtifact` block that contains the text of the Artifact. Notice that the `antArtifact` tag also includes attributes such as the title of the Artifact and the language used.

Table 6-3. The Anthropic structured prompt used for generating Artifacts (abridged)

Prompt	<p>The assistant can create and reference Artifacts during conversations. Artifacts are for substantial, self-contained content that users might modify or reuse, and they are displayed in a separate UI window for clarity.</p> <p>Here are some examples of correct usage of Artifacts by other AI assistants:</p> <pre><examples> <example> <user_query>Can you help me create a Python script to calculate the factorial of a number?</user_query> <assistant_response> Sure! Here's a Python script that calculates the factorial of a number: <antThinking>Creating a Python script to calculate factorials meets the criteria for a good Artifact. Therefore, I'm creating a new Artifact. </antThinking> <antArtifact identifier="factorial-script" type="application/vnd.ant.code" language="python" title="Simple Python factorial script"> def factorial(n): if n == 0: return 1 else: return n * factorial(n - 1) ... </assistant_response> </example> [...several examples omitted...] </examples></pre> <p>The assistant should always take care to not produce Artifacts that would be highly hazardous to human health or wellbeing if misused, even if is asked to produce them for seemingly benign reasons.</p> <p></artifacts_info></p> <p>Claude is now being connected with a human.</p> <p><user_query>Can you help me create a Python script to factor a number into its prime factors?</user_query></p> <p><assistant_response></p>
--------	--

Completion	<pre> Sure! Here's a Python script that factors a number into its prime factors: <antThinking>This code is self-contained and is likely to be reused or modified. Therefore, I'm creating a new Artifact.</antThinking> <antArtifact identifier="prime-factor-script" type="application/vnd.ant.code" language="python" title="Python script for factoring numbers into prime factors"> def prime_factors(n): factors = [] d = 2 while n > 1: while n % d == 0: factors.append(d) n /= d d += 1 return factors print(prime_factors(24)) # Output: [2, 2, 2, 3] </assistant_response></pre>
------------	--

As a result of setting up this structured pattern in the prompt, it becomes much easier to parse out information from the response. In particular, `antThinking` is parsed out and hidden from the user, and `antArtifact` is extracted and placed into the Artifact pane, under the title specified in the attributes. (In [Chapter 7](#), we get into much more detail about how content can be extracted from the completions.)

Like conversation transcripts, structured documents can come in many different formats. The Little Red Riding Hood principle suggests that you use formats that are readily available in training data. The most suitable formats are XML and YAML. Both are common in technical documents where precision is of the essence, and both can be used in many different domains. In both cases, the whole document is hierarchically ordered into normally named elements, which can have several subelements.

In [XML](#) (see [Table 6-3](#)), the document consists of a series of tags that are opened and closed. The tag may have attributes and has content that may contain subtags. Choose XML if your individual elements are relatively short, and if they are multiline, indentation doesn't matter. But you might need to be careful about escape sequences: there are five in XML: `"` ("), `&apos` ('), `<` (<), `>` (>), and `&` (&). XML also allows you to add HTML-style comments as `<!-- this is a comment -->`, which can occasionally be useful for "editorial" hints for the model.

In [YAML](#), the document consists of a series of named fields or unnamed bullet points whose hierarchy levels are tracked by their indentation. This indentation tracking can be quite annoying because you need to get it right to be able to use standard parsers, but it's helpful in cases where you need to be very precise about indentation,

such as with code or formatted text. In particular, the syntax `fieldname: |2` opens a multiline text field that preserves indentation, as illustrated in [Figure 6-4](#). Note that in such text fields, you don't need to escape anything, which is nice. You'll notice that such a text field is finished by encountering a line with smaller indentation than the text field's "zero" indentation. Also note that the highlight boxes indicate the value of the content fields, including leading whitespace.

The screenshot shows a portion of a YAML configuration file. It starts with `format: yaml` and `examples:`. The first example under `examples:` is a `singleline` type with content `a single line, duh`, which is highlighted with a red box. The second example is a `multiline text field` with content `|
the indentation of the first line
is the "zero" indentation`, where the entire line is highlighted. The third example is a `multiline text field with fixed indent` with content `|2
the "zero" indentation is
the indentation of the previous line
plus 2`, also highlighted with a red box.

```
format: yaml
examples:
  - type: singleline
    content: a single line, duh
  - type: multiline text field
    content: |
      the indentation of the first line
      is the "zero" indentation
  - type: multiline text field with fixed
    indent
    content: |2
      the "zero" indentation is
      the indentation of the previous line
      plus 2
```

Figure 6-4. Text fields specifying indented content in YAML

Another markup language that should feature heavily in any LLM's training set is JSON (or its variant, JSON Lines). At one point, we would have recommended against using JSON since it is very escape heavy and less readable. However, OpenAI in particular has put a lot of effort into making its models generate JSON accurately because JSON powers their tools API. Therefore, for OpenAI at least, JSON is still a reasonably good choice.

Formatting Snippets

The way you format snippet text depends a lot on your document. In an advice conversation transcript, you can format snippet information into the back-and-forth turns of the conversation. For instance, say your application retrieves this weather forecast data:

```
weather = {
    "description": "sunny",
    "temperature": 75
}
```

That information can be packaged as the advice seeker asking a clarifying question and the assistant answering with the following information:

```
User: What's the weather like?
Assistant: It's going to be {{ weather["description"] }} with a temperature of
{{ weather["temperature"] }} degrees.
```

In an analytic report, you normally want to state your knowledge in natural language. Results of API calls require you to know what the API returns, and then you can format the string into a sentence. Often, it's useful to include the results of individual API calls as individual sections, like this:

```
#### Weather Forecast
{{ weather["description"] }} with a temperature of {{ weather["temperature"] }}
degrees
```

Finally, if you use a structured document, your life is often easy: just serialize all relevant fields of the object you have in memory that represents your piece of knowledge:

```
<weather>
<description>sunny</description>
<temperature>75</temperature>
</weather>
```

No matter what type of document you use, a useful form of communicating background context can be a pretty explicitly stated side remark (e.g., “As an aside, ...”). For example, in GitHub Copilot code completions, where our document template was in the form of a source code file, we found we could usefully include code from other files using a code comment stating explicitly that some quoted snippet was included for comparison reasons, like so:

```
// <consider this snippet from ../skill.go>
// type Skill interface {
//     Execute(data []byte) (refs, error)
// }
// </end snippet>
```

An aside provides a strong hint to the model, but without requiring it to use the side remark in a certain way or at all.

When formatting your snippets, the things to aim for are as follows:

Modularity

You want your snippets to be strings that can be inserted into or removed from the prompt with relative ease. Ideally, your document is like a list (a conversation with turns) or a tree (a report with hierarchical sections; a structured document), so that snippets are easier to handle as items in the list or leaves of the tree.

Naturalness

The snippet should feel like an organic part of your document and be formatted as such. If you’re letting the LLM complete source code, any natural language information should be formatted as a comment rather than dumped between the code lines verbatim. If your document template is a conversation or a report, then data should be interpolated into a natural text that sounds appropriate for the document (see the preceding weather examples).

Brevity

If you can communicate relevant context with fewer tokens, great!

Inertness

You’d like to compute the token length of a snippet only once, so the tokenization of one snippet shouldn’t affect the tokenization of the previous or next snippet.

More on Inertness

The last bit, inertness, depends on your tokenizer, which might use different tokens to tokenize a composite string A + B than to tokenize each string individually. That can easily increase or decrease the number of tokens needed to tokenize a composite string (see [Table 6-4](#)).

Pasting strings together doesn’t mean the arrays of tokens just get concatenated. Token IDs have been obtained for OpenAI’s [GPT-3.5-and-later tokenizer](#), but both examples also work for the [GPT-3-and-before tokenizer](#) used in many non-OpenAI LLMs.

Table 6-4. Token count isn’t additive

	Example 1	Example 2
Strings	“be” + “am” → “beam”	“cat” + “tail” → “cattail”
Tokens	[be] + [am] → [beam]	[cat] + [tail] → [c], [att], [ail]
Token ids	1395 + 309 → 54971	4719 + 14928 → 66, 1617, 607
Token count	1 + 1 → 1	1 + 1 → 3

It’s generally a good idea to separate individual prompt elements with whitespace to prevent them from merging unexpectedly. However, be aware of potential issues:

GPT tokenizers often include tokens that start with a blank space but not ones that end with it. To avoid problems, prefer prompt elements that start with a space rather than ending with one. Additionally, GPT tokenizers combine multiple newline characters, so it's best to ensure that your snippets either never start or never end with a newline. Avoiding newlines at the beginning of snippets is usually easier for app developers.

Formatting Few-Shot Examples

When formatting snippets for few-shot examples, you usually have a choice. One option is to designate them explicitly as examples, as shown here:

In the following, when I encounter a question like "Who was the first President of the United States?" I will give an answer like "George Washington."

Alternatively, you can integrate examples directly into the document as solutions to previous tasks. This approach requires careful formulation but can be highly effective. It allows the model to leverage the few-shot examples more naturally and creates a smoother prompt. This method is especially useful in ChatML or similar conversation transcript settings, where you can make the model believe it has successfully solved previous tasks in the style of the examples, thus encouraging it to continue using that successful approach.

Elastic Snippets

When you're converting content into snippets, each piece of information usually corresponds to a single snippet. However, sometimes, one piece of content can be split into multiple snippets or be represented in various forms.

For example, consider a literary analysis task asking about the significance of a specific scene in Alex Garland's novel *The Beach*. If you ask ChatGPT about this scene, chances are that it won't be familiar with that particular one, and any **answer** it gives will be vague, wrong, or both. To improve the response, you need to include relevant context from the book in your prompt. You remember how to retrieve relevant book passages from [Chapter 5](#), and let's say you identify two key moments.

You can snippetize these passages in different ways, as shown in [Figure 6-5](#). Ideally, you'd include the entire chapter for full context. That's one way, but with limited prompt space and limited model attention, you probably must tighten your belt regarding context a bit. But that leaves you with different possibilities:

- Add two snippets with no context around them.
- Add two snippets with some context around each.
- Add one combined snippet with context linking the parts.

All three options have things to recommend them. The first option is short, the last option conveys most of the information (including how the snippets relate to each other), and the middle option is somewhere in between. But of course, there are even more options: you can choose a tiny bit of context, a lot of context, etc. How do you deal with such a situation?

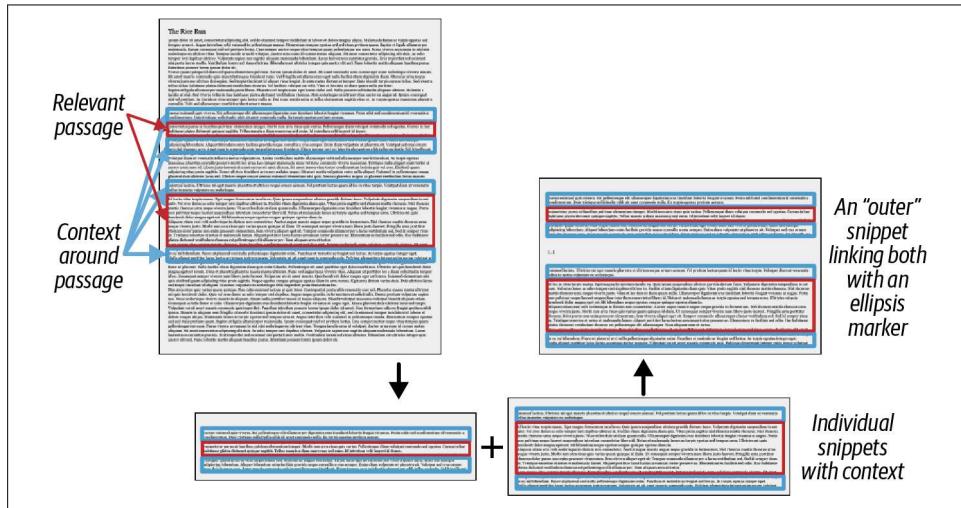


Figure 6-5. Snippetizing context into flexible snippets

There are two general approaches to such a situation where there's a variable amount of context you could include. We have used both, depending on our exact requirements:

1. You could use what we call *elastic* prompt elements, which are prompt elements that have different versions, ranging from short to long. In this case, the longest version would be the whole chapter, a slightly shorter version would be one where one paragraph is replaced by "...," and an even shorter version would be one where two paragraphs are replaced by "...". This can go all the way down to the shortest version, where it's just the two snippets you want to quote with no extra context and a "..." between them. Then, when you're assembling the prompt, you don't ask, "Do we have space to include this snippet?" Rather, you ask, "What's the biggest version of this snippet we have space for?"
2. Alternatively, you could create multiple prompt elements from the retrieved information. For example, you might have one snippet that is the first relevant text passage, another that is that passage plus some context, and another with even more context. You'll have to remember to only actually include one of these, as they overlap. So, this approach requires a prompt assembly method that allows you to declare prompt elements as incompatible (see the next section).

Relationships Among Prompt Elements

Prompt elements don't exist in a vacuum: a prompt is an amalgam of several of them. Any algorithm that combines prompt elements has to account for three ways in which the elements relate to each other: position and ordering, importance, and dependency. You'll need to keep these three dimensions in mind while you're constructing the prompt elements. Let's go through each of them.

Position

Position determines where each element should appear in the prompt. Prompt elements usually need to follow a specific order—while you might skip some, *rearranging* them can make the document confusing. For example, if you're quoting from reference documents, you should maintain the original order; don't place the second snippet before the first. In chats or narratives, stick to the chronological order. In other situations, ensure elements are in the correct sections; for instance, a description of a book the user likes shouldn't go in the “Books I really hate” section.

To manage these relationships, you might use an array or linked list of prompt elements, an index covering all elements, or a unique position value for each element. Often, the order reflects how you gather information (e.g., scanning a document or retrieving context section by section). In such cases, you generally only need to append new elements to the end.

Importance

Importance determines how crucial it is to include a prompt element to convey relevant information to the model. Beginners often confuse position with importance, as they are frequently correlated—recent information is often more important. But there are many exceptions—for example, your introduction is often more important than most of the details in the middle (which are rightfully consigned to the Valley of Meh from [Figure 6-1](#)).

When evaluating the importance of each element, consider the tradeoff between including large chunks of relevant information and including many smaller, less critical elements. Decide whether to measure importance based on snippet length or an absolute scale, but choose one method and apply it consistently. Short, efficient prompt elements are often preferable to longer ones that convey the same amount of information. If you don't account for length initially, ensure the prompt assembly engine can adjust importance based on token length later.

To assess importance, use either a numerical score or discrete priority tiers. *Tiers* are a small number of levels you can quickly sort your sources into, with lower tiers being cut first, if necessary. Some elements —such as the central instructions and the description of the output format—are so vital that they must be included at all

costs. These need to occupy the highest tier. Next typically come explanations in the second-highest tier and context in the third. But as you delve deeper into the subtleties and compare different context sources or different degrees of relevance, consider adding numbers for finer prioritization.

Assigning importance involves judgment and is crucial for effective prompt engineering. You also need to test and refine these importance parameters with the methods we'll explore further in [Chapter 10](#).

Dependency

Dependency is the final type of relationship among prompt elements, and it focuses on how including one element affects the inclusion of others. Dependencies can be complex, but in practice, they usually fall into two categories—requirements and incompatibilities:

Requirements

These occur when one prompt element depends on another. For example, you need to establish that “Richard is the protagonist of *The Beach*” before stating “He grew up in England.”

Incompatibilities

These occur when one prompt element excludes another. This often happens when the same information can be presented in different ways, such as in a summary versus a detailed explanation. If your prompt assembly engine can handle incompatibilities, you can include both versions with an exclusion note, giving the longer version when space allows and using the shorter version as a fallback.

At this point in the text, you should have transformed all your pieces of content: the static ones that you prepared beforehand and the dynamic ones that you gathered as context into proper prompt elements like the ones in [Figure 6-6](#). That means you’re finally ready to assemble your prompt.

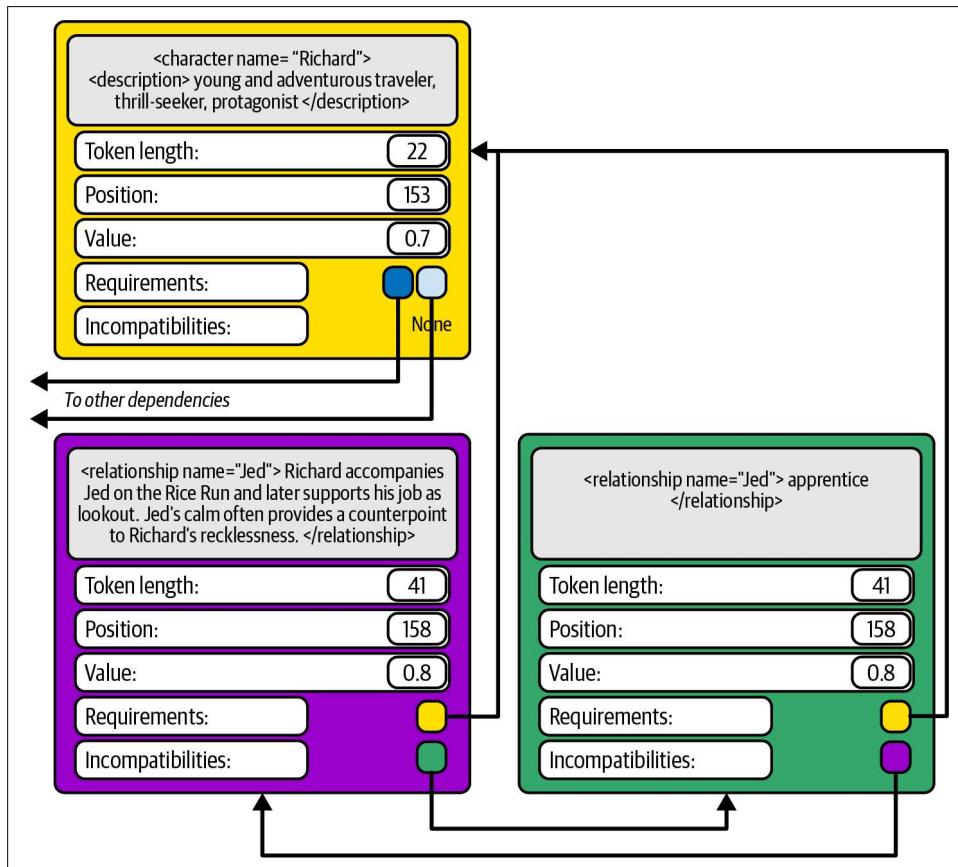


Figure 6-6. Prompt elements and their properties, including all the information you need to assemble your prompt

Putting It All Together

To create the final prompt, you need to solve an optimization problem: deciding which elements to include in the prompt to maximize its overall value.

You have two main constraints:

Dependency structure

Ensure that any requirements and incompatibilities between elements are respected.

Prompt length

Keep the total prompt length within a set limit, typically your context window size minus the tokens needed for the model's response. If your context window is

very large, you might use a softer token budget based on available compute and to avoid including too much irrelevant context.

Once you decide which elements to include, arrange them according to their positions to form the final prompt.

This problem is similar to linear programming and 0-1 knapsack problems, where you decide whether to include an element (though knapsack problems often don't account for dependencies). However, there isn't a standard tool that automatically solves this for you, so you'll need to create your own solution. It can be a rewarding process, allowing you to customize it to your specific needs.

Consider what you need from your prompt assembly; for example, if you need a fast assembly for interactive apps or if you have specific dependency patterns to handle. In Copilot code completions, code snippets often require a specific postfix, so we handle these with custom functions that manage dependencies between lines of code.

When developing your app iteratively—starting with a basic version and then expanding—it's useful to begin with a minimal prompt crafter like the one shown in [Figure 6-7](#). This simple tool helps you test whether your app idea has potential. With this approach, you don't need to evaluate or prioritize snippets because the prompt crafter uses only the end portion of your content. This method works well because LLMs are trained to handle document suffixes effectively. It's also suitable for applications where you build on a main text or for chatlike applications where recent exchanges are most relevant.

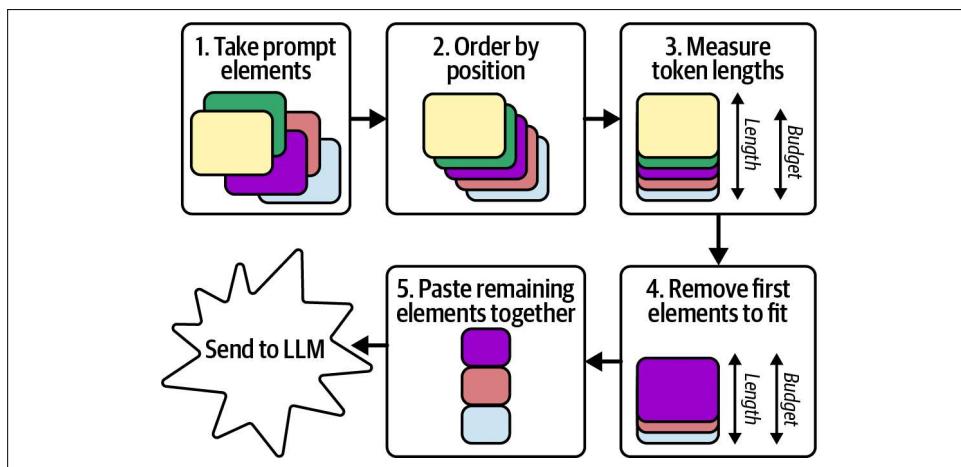


Figure 6-7. The minimal prompt crafter, which orders prompt elements and keeps as many at the end as it can fit into the token budget

As your app develops, you'll need a more advanced prompt-crafting engine. For speed, consider using a greedy algorithm as shown in [Figure 6-8](#) (possibly combined with some limited exploration of alternatives). There are two main types of greedy algorithms you can use, depending on how your prompt elements interact: an additive approach and a subtractive approach.

In the *additive greedy approach*, you start with an empty prompt and add elements one by one. Each step involves adding the highest-value element that meets all requirements, doesn't conflict with existing elements, and fits within the prompt length. This method is even effective if you have many more elements than fit into your prompt and you need to eliminate a lot. However, it requires few cyclical requirements and few cases of high-value elements dependent on low-value ones.

When using the additive greedy approach, you can simplify the process of finding the best element to add by sorting your elements based on their requirements and values. This way, you consider elements only once all their dependencies are satisfied.

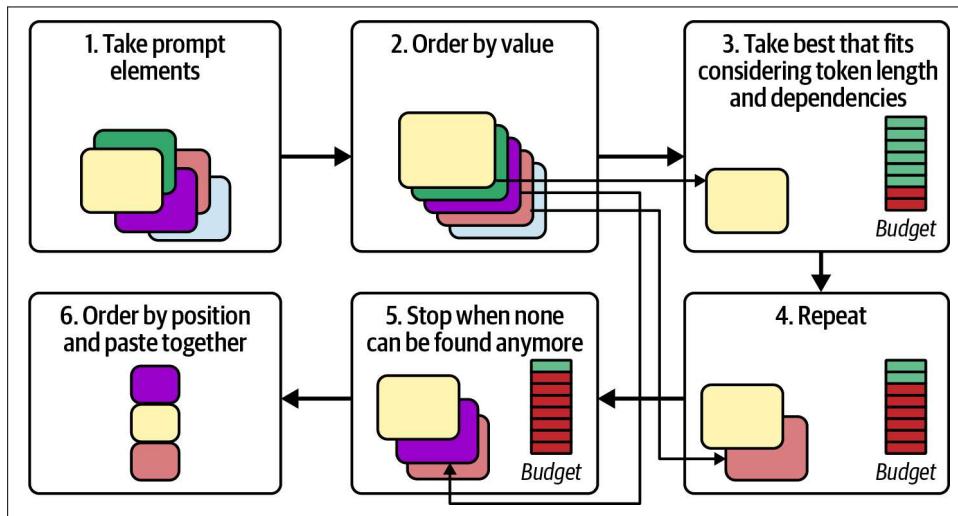


Figure 6-8. Additive greedy approach, in which the prompt crafter iteratively adds high-value elements to the prompt until the token budget is filled up, then re-sorts the elements according to position

With the *subtractive greedy approach* shown in [Figure 6-9](#), you start by including all prompt elements and then gradually removing those that are less valuable or whose dependencies are no longer met. This method works well if you have a manageable number of elements and few incompatibilities. Otherwise, the process can become cumbersome. High-value elements dependent on low-value elements can also lead to suboptimal results, unless you use advanced techniques to prioritize retaining

high-value dependencies. Elastic snippets are normally easier to handle in a subtractive approach than in an additive approach.

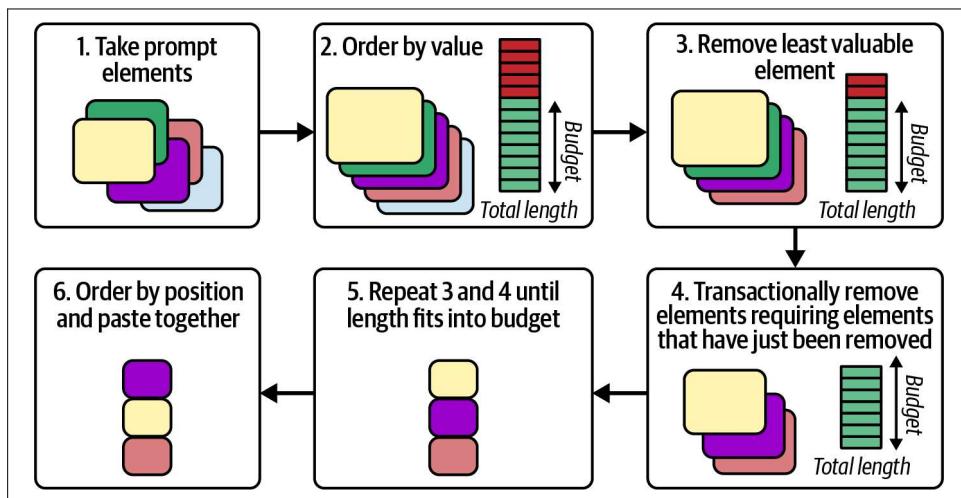


Figure 6-9. The subtractive greedy approach, in which the prompt-crafting engine successively eliminates prompt elements of low value, pruning missing requirements in between

Please note, however, that all of the sketches for prompt-crafting engines presented in this chapter are meant as basic prototypes. Maybe you'll find them sufficient for your application, but you should be willing to move beyond them in a way dictated by your specific requirements as these requirements become clear to you when you refine your application.

Conclusion

In this chapter, we've covered the art of crafting an effective prompt from your gathered information. We explored how to choose the right document format and examined various prototype documents that LLMs excel at completing.

You've also learned how to convert your information into prompt elements—snippets of text that fit seamlessly into your document while aligning in relevance, order, and dependency. Now, you'll be able to refine these elements to create a concise and effective prompt using a custom prompt-crafting engine, inspired by the strategies covered here, with ease.

Congratulations on completing the feedforward pass from [Chapter 3](#)—you've successfully created a coherent prompt for the model. In the next chapter, we'll focus on how to ensure that you receive meaningful and accurate responses.