

CHAPTER 7

Taming the Model

In the previous chapter, you managed to distill all your context into a single, coherent prompt. Now, it's time for the LLM to do its thing and for you to make sure that it all goes smoothly.

In this chapter, we're going to start by talking about completion formats and making sure your completions stop when they're supposed to, as well as how to interpret them using so-called *logprob tricks*.

Then, we're going to take a step back so you can ask yourself which model you're going to choose to invoke: a professional commercial service, an open source alternative, or even your own bespoke fine-tuned model. Time to get into it.

Anatomy of the Ideal Completion

In this section, we'll examine how completions appear, whether they're classic completions or chat responses. More importantly, we'll discuss how you want them to look to ensure clear and effective solutions, all while avoiding issues like unnecessary delays or confusing details. As we did in [Chapter 6](#) with prompts, we'll break down the components of an LLM completion and go through them one by one (see [Figure 7-1](#)).

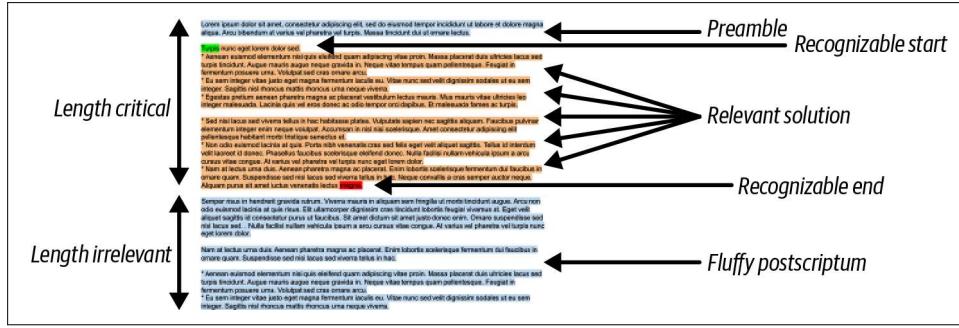


Figure 7-1. An LLM completion

The Preamble

In the context of completions, the *preamble* is the initial part of the generated text that sets the stage for the main content. Sometimes, this is helpful, and sometimes, it leads to completions that start with uninteresting or useless detail before they produce a solution to the problem you posed. This is often annoying, and it's costly too: generating tokens costs time (latency) and compute (resources and money). So, producing text that you're not going to use is wasteful, but sometimes, it's desirable. We know, it's confusing, but stay with us here.

Whether it really *is* wasteful or whether you can avoid it depends on the exact type of preamble. There are three different types of preambles, and we'll explore what each of them:

Structural boilerplate

This is the text between the end of a prompt and the start of a completion. When using a completion model, you might be able to eliminate this preamble type, but it's more efficient to include deterministic boilerplate in the prompt rather than the completion, thus ensuring that the model adheres to the desired format and making the process faster and cheaper. Structural boilerplate makes for a good transition from the prompt to the completion.

Reasoning

Toward the end of 2023, ChatGPT started mirroring a slightly interpreted version of questions to clarify understanding and highlight potential misunderstandings. This approach helps the model make better inferences by focusing on key aspects of the prompt and ensures more accurate responses. Additionally, chain-of-thought prompting, as discussed in [Chapter 4](#), helps the model break down problems into manageable pieces, with the detailed process often being part of the preamble rather than the main answer. If you're doing chain-of-thought prompting, having a long preamble is a virtue, not a vice, even if it's significantly longer than the actual answer (see the example in [Figure 7-2](#)). Also note that

in the figure, the answer arrived at after a long preamble is correct, while the answer arrived at after a short preamble is not. Many of the advanced prompting techniques discussed in [Chapter 8](#) will center on making good use of reasoning preambles as well.

User: I'm thinking of a number between 1 and 10. It's not a prime. It's not the power of a prime. What number might I be thinking of?
Please make a list where for all numbers between 1 and 10, you compute its prime factorisation, and whether it's possible or eliminated.
Then write {a₁, ..., a_n} where the a_i are exactly the possible answers.

ChatGPT: Sure, let's go through the numbers from 1 to 10 and eliminate those that are primes or powers of primes:
1. 1: Not a prime, but $1^1 = 1$ (power of a prime)
Eliminated.
[...]
10. 10: $2 \times 5 = 10$:
Possible.
Therefore, the possible numbers are {6, 10}, as these are not primes and not powers of primes between 1 and 10.

Versus

User: I'm thinking of a number between 1 and 10. It's not a prime. It's not the power of a prime. What number might I be thinking of?
Please start by writing {a₁, ..., a_n} where the a_i are exactly the possible answers.

ChatGPT: The possible answers, given your criteria, are {4, 6, 8, 9}.

Figure 7-2. Encouraging long preambles to get a correct answer

Fluff

RLHF-trained models often produce verbose and polite responses, which can be problematic for programmatic use where succinct outputs are needed. While models with RLHF are prone to including unnecessary fluff, even those without it can occasionally produce it. To manage this, you can use techniques like providing instructions with few-shot examples or reformatting prompts to separate the main answer from additional comments. This can be expensive, though. For structured documents, models generally maintain the format, but for free-form

contexts, asking for the main answer first followed by any extra information helps in parsing and reducing the impact of fluff.

Which portions of fluff to reserve depends on what kind of fluff the model you chose tends to supply for the kind of questions your application asks. Typical candidates are comments, disclaimers, background, and explanation (see [Figure 7-4](#)). Note that the point of this figure is not to demonstrate a correct answer, but to demonstrate the format. Also note that while this trick is good at banishing most fluff behind the main answer, it will not always get rid of a short introduction before the first numbered list item (see [Figure 7-3](#)).

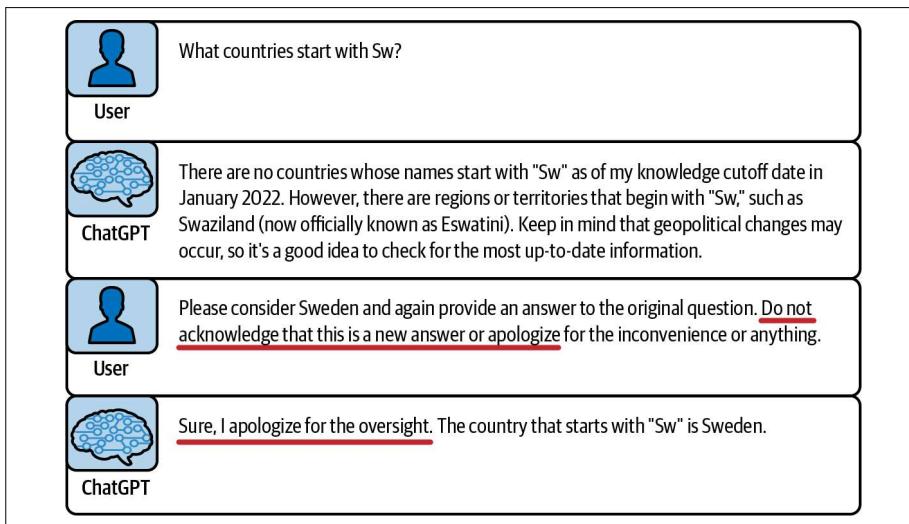


Figure 7-3. A fluff preamble that ChatGPT included against explicit instructions in its second answer

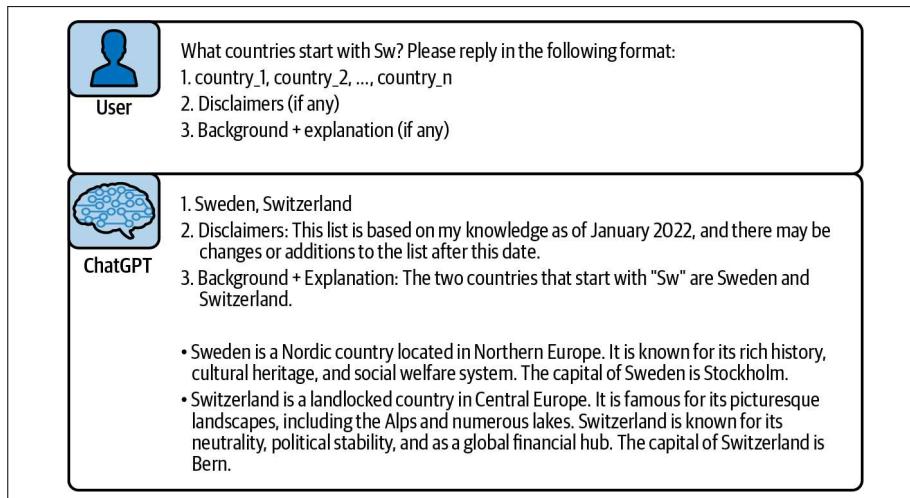


Figure 7-4. Banishing all of ChatGPT's fluff into a subsequent point, so it can be parsed out easily

Recognizable Start and End

If you want to pick out your main answer from the LLM's response, you must be able to recognize the beginning and the end. Many document structures make this relatively easy (see [Table 7-1](#)).

Table 7-1. Recognizable start and end examples and whether the test for the recognizable end can be written as a test for the presence of a substring

Document structure	Start	End	Test for end is test for substring
A Markdown document	The expected section header	Any other section header	Yes
A YAML document	The expected keyword after a newline	A line with lower indentation	No
A JSON document	The expected keyword in quotation marks, then a colon and a quotation mark	Any unescaped quotation mark	No
A triple-ticked (``` code listing	```[language]\n	\n``` \n	Yes
The first item of a numbered list (see comments about fluff)	1.	2.	Yes
A function/class in source code (a bracketed language like Java)	{	The matching closing bracket	No
A function/class in source code (an indent language like Python)	The expected function/class header	A lower indentation level (except for the occasional terrible string literal)	No

As [Table 7-1](#) shows, figuring out the start and end of a section can either be straightforward or a bit tricky. With a well-crafted prompt, you can sometimes improve on the recognition methods shown in the table. For example, in a YAML document, if you know what the next keyword will be, you can look for a lower indentation level followed by that keyword, rather than just any lower indentation. This means you can determine the end by checking for specific substrings, as described in the fourth column of [Table 7-1](#). Next, we talk about identifying the end of the main answer.

Postscript

The reason why your start should be recognizable is clear: it helps you when parsing the answer to filter out the irrelevant introduction. As with the end, you want to be able to filter out the fluffy postscript that's not relevant to your question.

But there's a second, at least equally important consideration that applies here. You want to be able to control the length of the LLM's answer. Every generated token costs you time and compute, making your application slower and more expensive. So ideally, you want to stop generating tokens whenever you hit your recognizable end. If you're self-hosting an OS model, you have complete freedom to do that whenever you want. But it's much more common to call an existing model as a service. Here are the two main ways to do it:

Stop sequences

Many models, in particular those following the OpenAI API, allow you to provide a *stop argument* that's a list of sequences you know mark the end of the relevant solution. When it reaches one of those stop sequences, the model generation will stop (on the server side, if it's on a server) and end its answer. You won't incur any further cost in waiting time, compute, or money.

Streaming

Several models allow a *streaming mode* where either individual tokens or small batches of tokens are sent one at a time, instead of waiting until the model generation is complete. Models following the OpenAI API activate streaming by setting the "stream" parameter to "true." Recognizing an end while streaming means you don't have to wait for the generation of additional, uninteresting tokens. If you cancel the generation (and the model supports that), you can even save yourself some compute and money—but not as much as you'd have saved with stop sequences, because network communication delays mean your cancellation signal won't get through immediately.



Very often, stop sequences will begin with a newline character. For example, in markdown documents, `\n#` is a typical stop sequence. If you don't include the newline, then you may erroneously stop on a comment in code or the beginning of a phone number.

Typically, more models admit stop sequences than allow streaming and cancellation, and stop sequences are a tiny bit more effective. But since stop sequences are limited to a list of specific strings, sometimes canceling streams is the only viable option.



If certain sequences occasionally signal the end of the completion, you can enhance your “streaming and cancellation” method by adding them as stop sequences. For example, when generating a Python class, `\nclass`, `\ndef`, and `\nif` are such sequences. They are not the only way the code can continue after the class, but they are some of the most common ways. You might think that `\ndef` is incorrect because the class you’re generating will have several methods defined that start with `def`, but notice that they will be indented and will actually start with `\n\def`. Therefore, they will not cause the model to halt generation.

Beyond the Text: Logprobs

Throughout this book, we’ve been presenting LLMs as “text in” (prompt) then “text out” (completion). But it’s worth being aware of a couple of tricks that break that paradigm by analyzing not only the text output but the numerical values that describe what the model thinks about the text.

In [Chapter 2](#), we discussed how the LLM calculates not just individual tokens but the entire probability distribution for the next token based on previous input. These probabilities are returned as *logprobs* (the logarithm of the probabilities). A logprob is negative; the more negative its value, the less probable the token is considered by the model. A logprob of 0 means the model is certain about the token. To convert a logprob to a standard probability, you use the `exp` function. For instance, if the logprobs for “Yes” and “No” are -0.405 and -1.099, respectively, then the model is about 66% sure it will be “Yes” and 33% sure it will be “No.”

For models using the OpenAI API, you can request that those logprobs be returned to you as shown in [Figure 2-12](#). What you get are the calculated probabilities, not just for the tokens that the model ends up choosing, but also for the ones it considered and decided not to use. Since the model calculates these probabilities anyway, retrieving them doesn’t require any additional computing effort.



Some commercial models disable the part of the API where you get the logprobs, mostly out of fear of being reverse-engineered if they share too much about their internals. If you want to use any of the tricks in this section, consider this in your LLM choice.

You can do many cool things with logprobs. Let's talk about how to use them to evaluate answer quality, get the model to estimate certainties, and find critical locations in a (provided or generated) text.

How Good Is the Completion?

Albert's neighbor happens to be an astrophysicist, and when Albert asked her how many minutes light needs to travel from the sun to Mars, roughly; she straightaway replied, "13," with absolute confidence. Albert then asked his 10-year-old daughter the same question. She looked surprised and then hesitantly guessed, "Maybe 30?" One of these answers is much more reliable than the other, and anyone present can tell which one is more reliable from the respondent's facial expressions and tone of voice. Well, logprobs are like the model's tone of voice, and you can use them to see how confident it is in its answer—and that's a strong indicator of answer quality.

Logprobs indicate a model's confidence in each token choice (refer to [Figure 7-5](#)). Summing logprobs across a text shows overall confidence in that text as the "correct" response, considering how it might start with the prompt in training data and conclude with the completion. However, the accuracy of this measure can decrease with longer texts due to the many ways in which the same idea can be expressed, like using "for example" or "for instance," which can halve the probability without reflecting a decrease in quality.

To assess quality, it's beneficial to average the logprobs. The simple average—adding all logprobs and dividing by the number of tokens—is effective, especially if experimenting isn't feasible due to constraints like data scarcity or limited time. For a more nuanced approach, Albert, during GitHub Copilot's development, found that averaging the probabilities (rather than the logprobs) of early tokens in the completion is predictive of overall quality. (This is calculated as `(exp(logprob_1) + ... + exp(logprob_n)) / n.`)

This average provides a numerical quality indicator, and while it falls short of being an absolute measure of quality, in practical applications, you can explore logprob-based cutoffs for features within your application as follows:

1. Only allow your application to show corrections if it is confident.
2. Include warnings when the model struggles more than usual.
3. Incorporate more context or retry when the model struggles.
4. Switch to a more intelligent (and expensive) LLM for better results.
5. Only interrupt the user with assistance if the certainty that it is necessary is high.
Remember [Clippy](#)? Don't be like Clippy.

For greater quality at a higher compute cost, you can also consider setting a higher temperature, generating multiple completions, and choosing the best one based on their logprobs.



Many LLM APIs have a parameter called n , which controls the number of completions that are generated from the same prompt in parallel. If n is larger than 1, then the temperature should be larger than 0 or all completions will be the same. A rough (and completely unscientific) rule of thumb we like to use is temperature = $\text{sqrt}(n) / 10$.

LLMs for Classification

The concepts of classification and logprobs are intertwined in the context of LLMs, as logprobs provide critical insights into the model's decision-making processes, confidence, and reliability. Let's take a look at classification now.

Classification is a basic machine learning task in which you determine which category a specific case belongs to from a set of predefined options. For instance, you might classify an online review as positive, negative, or neutral, or you could predict whether a product is best suited for the American, European, or Asian market. In simpler terms, you could be deciding if the answer to a question is yes or no. The key aspect is that, much as in a detective novel with a limited number of suspects, there are a fixed number of possible categories, and your goal is to identify the correct one and determine your confidence level in that choice.

This is pretty much the opposite of how LLMs were built to work: LLMs lean toward long, creative generation instead of fixed, boxed-in classification. But LLMs are pre-trained generalists, and in domains where the classification task relies on public knowledge and common sense, they have a good chance to excel with little to no extra training data. The prompt engineer has to set up the prompt in a way that the model chooses exactly one of the alternatives. But there are some subtleties, which we'll talk about now.

At the basic level, you use your LLM just by asking it questions. If you want to find out whether a sentence is positive, negative, or neutral, you might present the sentence to the model and add the question, "Does that sound positive, negative or neutral to you?" Then, you might check the answer for which of these three alternatives occurs in it. Of course, you want to avoid waffling answers that include several alternatives, like "more positive than neutral." A more refined question might be "Does that sound positive, negative, or neutral to you? Please answer in the format: 1. [negative | positive | neutral], 2. [explanation]." The "1." in this example is what we called a *recognizable start*, and you can expect the answer directly after it.

In this situation, it's a good idea to ensure that after the first recognizable token, you can immediately tell which option the model chooses. Here's why: in [Figure 7-5](#), the model has three options: North America, Northeast Asia, and Europe. Two of these, North America and Northeast Asia, both start with the token *North*. When the model predicts the next token, the two answers that start with *North* combine their chances, since the model predicts only *North* at first. If the model is uncertain, it's more likely to choose *North* because both options share it. The actual decision between the two will come afterward. To avoid this, you need to make sure each option starts with a unique token.

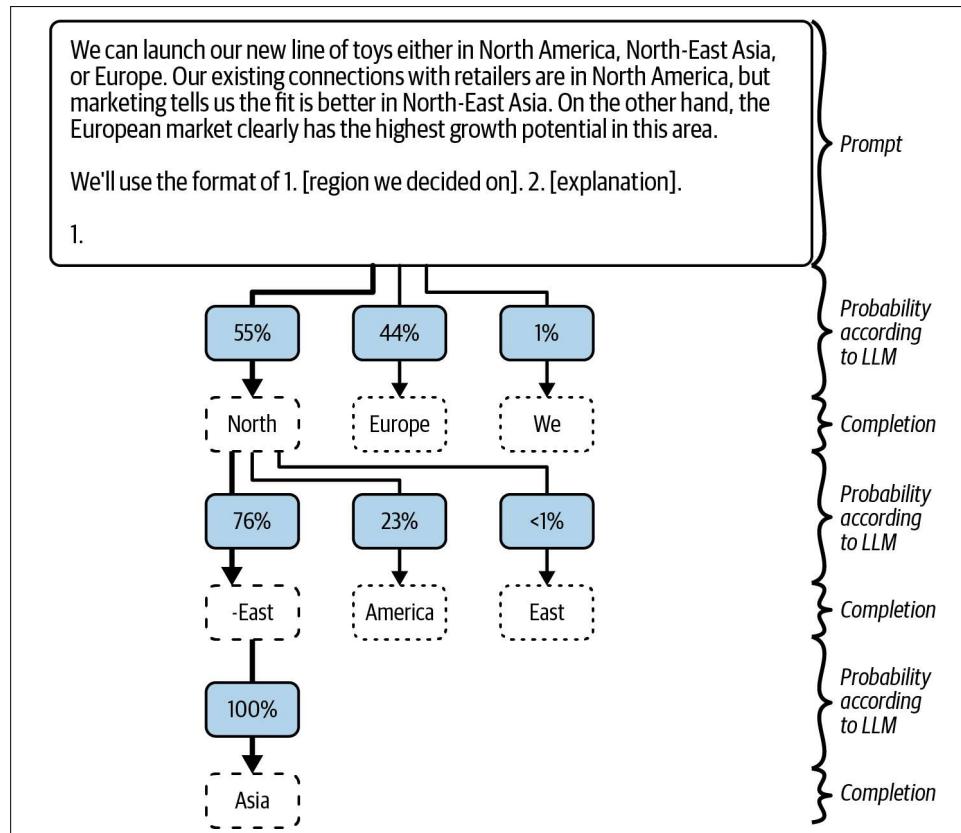


Figure 7-5. The model's calculated total probability for Europe is highest (44% versus $55\% \times 76\% = 42\%$ for Northeast Asia), but the suggestion will be Northeast Asia

Note that in the figure, because the first decision is between North and Europe, the probabilities for Northeast Asia and North America are added together, leading the model to put out a suggestion it actually considers suboptimal. These are actual probabilities from OpenAI's gpt-3.5-turbo-instruct.

You can let the model make all kinds of decisions through classification, but in many situations, its prediction will be badly calibrated compared to what you want. For example, let's say you're writing an app that helps grumpy users by blocking some of the emails they write if the LLM deems them not sufficiently friendly and asks them to rewrite them. You can pretty easily ask the model, "Is this a professionally written email? Please use the format 1. Yes / No. 2. Explanation." But even if the model is good at recognizing whether one email is more professionally written than another, the threshold between what you do and don't consider to be professional is likely not the same as the model's. To match the model's threshold more closely, you'll have to calibrate, and that's where the logprobs finally come into play.

Calibration means adjusting the certainty of a classification to better match the "true" certainty. A priori, the certainty of the prediction is the logprob, and whatever token has the highest logprob is what the model will produce (at temperature 0). But if, for example, you find that the model lets through too few emails, you'll wish that the model would only output No if it's super certain. So maybe it should only choose No if the logprob for No is at least 0.3 higher than the one for Yes.

Generally, to calibrate the LLM's decision process, you shift the logprobs by a constant (where each a_{tok} corresponds to one of the tokens in question). For example, you can make the email classification less strict by adding a constant like $a_{yes} = 0.3$ to the logprob of "Yes" before comparing it with the logprob for *No*. You can find these constants either by experimentation or by some classical machine learning: taking ground truth data and minimizing the [cross entropy loss](#) like you do in [logistic regression](#).



If you found constants a_{tok} that you like, you don't actually have to mess with the logprobs anymore—many model providers offer in their API the possibility of a *logit bias*, where you send the a_{tok} to the model and they will be applied for you.

Critical Points in the Prompt

Another application of logprobs isn't to get the certainty in the completion, but to understand the surprising parts of the prompt. Setting the parameter "echo" to true tells many APIs to return not only the logprobs for the completion, but also for the prompt. You can run this to better understand the text you send to the model, even if you don't request a single completion token.

For example, in the paragraph you just read, did you notice a typo? So did the model. As shown in [Figure 7-6](#), when you're displaying the logprobs, that typo stands out like a sore thumb with a logprob of below—13, where instead of the "completion" token, the model got only the "compl" token (followed by "ution"). This way, you can use logprobs to detect not only typos but also otherwise surprising parts of

the text. More generally, you can use logprobs to detect passages in the text with higher information density, with the idea of focusing your app’s attention on certain locations or alternatively guiding the user’s attention.

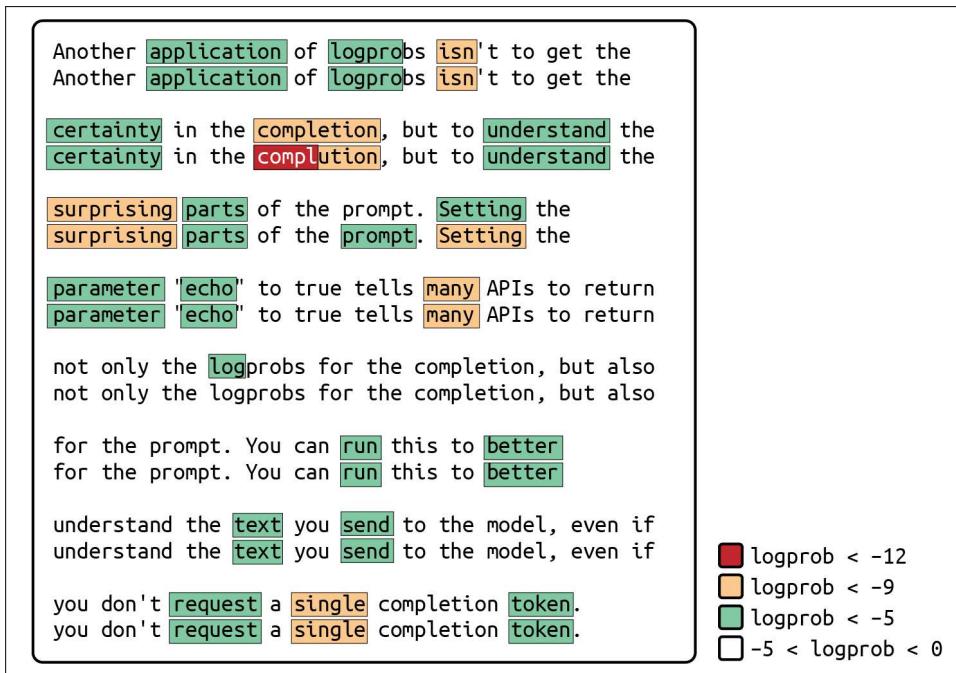


Figure 7-6. Logprobs of two versions of a paragraph of text, shown interleaved

As you can see in Figure 7-6, negative single-digit logprobs are somewhat common, while negative double-digit logprobs are usually the model picking up some weirdness. However, there’s no clearly delineated threshold, and even heuristics vary from model to model and genre of text to genre of text. In fact, they vary *within* a single text: in the beginning, the logprobs are usually lower (i.e., further below 0) than toward the end. That’s because much about the topic and style of the text becomes clear to the model only as it’s reading it.



When writing unit tests for any parts of your application that deal with logprobs, remember that due to floating-point inaccuracies, logprobs are not deterministic. Depending on the model deployment, they may vary by as much as ± 1 , so write your tests to be robust against such variation or mock out the model entirely.

Choosing the Model

In this chapter thus far, we've focused on the model itself, but we've danced around an important question: which model should you use? LLM choice is going to be critical to the success of any AI software development project, and yet, there are many alternatives, with new ones popping up every week. In a landscape that's changing this quickly, recommendations for particular models are going to become stale pretty quickly, so we'll instead focus on the underlying principles that should guide your choice.



Whatever model you end up choosing, don't bake your choice into your code too firmly. You may want to revise, evaluate, and refine your choice. Libraries like [LiteLLM](#) may be useful here for providing a unified API to many different models.

The model you *need* depends on what you *want*. There's no single quality that reigns supreme, but here's a list of of considerations (in order of importance) for most scenarios:

Intelligence

How close is the model's answer to that of an intelligent human expert with strong subject matter expertise? This is especially important for apps that ask the model complicated questions that require complex reasoning or very accurate answers.

Speed

How long do you have to wait for your answer? This is especially important for apps that interact very directly with their users (see [Table 5-2](#) about the different levels of urgency users may feel depending on the kind of application).

Cost

How much do you pay for running inference, either directly to the model provider or in costs for GPUs? This is especially important for apps that make very frequent requests to the model.

Ease of use

How much of the work regarding arranging GPUs, deploying the model, restarting crashed instances, routing, caching, etc., is conveniently done for you?

Functionality

Does the model have the capabilities for instruct, chat, and tool use? Does it surface logprobs? Can it process images as well as language?

Special requirements

These are a lot like dietary requirements; for some people, they are nonnegotiable, but for others, they are completely unimportant. Some app developers might prefer models to be noncommercial, open source, trained on specific data, and regularly updated (or not). They might want to ensure data residency in a particular country, or they may avoid logging off premises. These preferences can quickly narrow down the available options (see Figure 7-7).

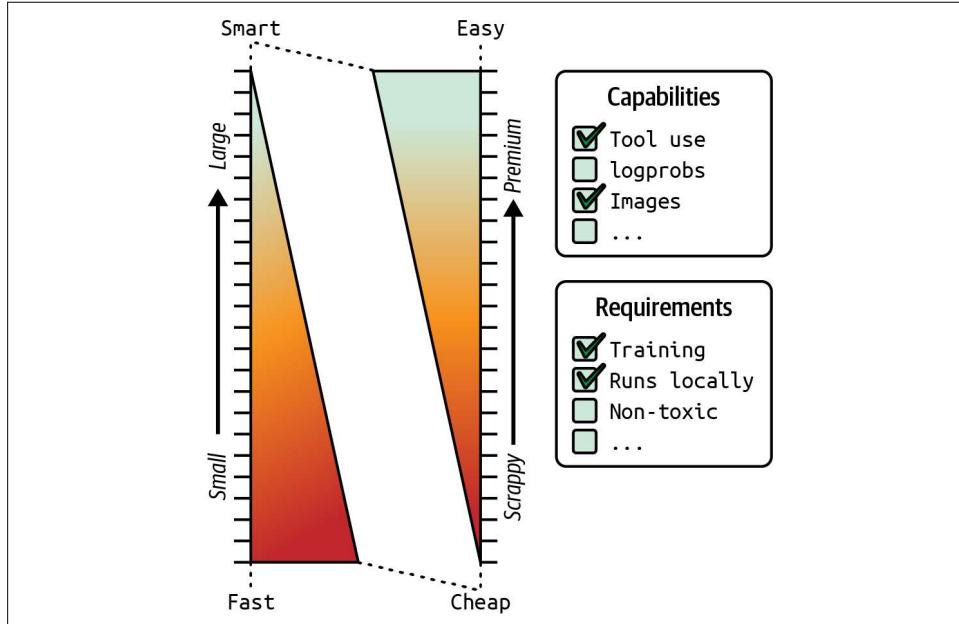


Figure 7-7. The knobs and dials you use to decide what kind of model you'll have

As Figure 7-7 illustrates, as you tighten one requirement, you often constrain the type of model available. Here are some examples:

- If you know that your app will make a high volume of relatively simple requests to the model and that you'll therefore need it to be cheap but not smart, a small model is likely to be appropriate.
- If your app is a solo project that you want to knock out quickly, and if it only makes about one request per day, then you should feel encouraged to splurge on a premium-tier model because cost may only be a factor at scale.
- If you make a ton of very difficult requests and need your model to be super cheap while being super smart...tough luck, because these two qualities are at opposite ends of the spectrum.

When you’re deciding on a model, the first step is usually picking a provider. You’ll likely base this decision on your requirements, desired features, and whether you want a scrappy or premium solution. Most providers offer a range of models, and you’ll narrow them down based on specific capabilities and needs, and then you’ll choose the model size.

At one point, OpenAI, known for its highly advanced models and full-service platform, was the dominant choice. However, over the course of 2024, the playing field has become more even. Here are some other choices to consider:

Anthropic

Emphasizes human alignment and AI safety. Its Claude 3.5 Sonnet model recently (in 2024) jumped to the top of several LLM benchmarks (see [Claude 3.5 Sonnet’s website](#)).

Mistral

Specializes in highly efficient, open-weight models; ideal for applications that need very specialized configurations.

Cohere

Popular for high-performance RAG applications.

Google

Strong integration with Google’s ecosystem, cutting-edge research, and large-scale infrastructure.

Meta

Large, highly capable open-access models.



There are many model comparison sites around to serve as a starting point in your exploration of what to start prototyping with or which alternatives to evaluate in more detail. We quite like [the Artificial Analysis website](#).

But if having the premium tier isn’t a requirement for you, then you don’t have to rely on an LLM-as-a-service company at all. Several LLMs like LLaMA and Mistral are open source and typically trained by academic groups or open source-friendly companies. Hosting these models requires significant effort, though platforms like Hugging Face aim to ease the process, whether you use your own servers or their Azure partnership. We recommend this route only if your app is large enough to justify the infrastructure investment and if your model needs to steer you away from full-service solutions. If you’re using agile methods, you can also prototype using the easily accessible OpenAI APIs with the intention of moving to a different platform when going public.

After you've found a provider, you'll probably have to choose among several different models the provider offers, and apart from some consideration of capabilities, this mainly means choosing the model size. Whether or not latency matters, completion quality versus cost is always a hard trade-off. Typically, you'll want the smallest model that can reliably deliver on your task.



Feel free to prototype with slightly larger models than you think you can afford. As new flagship models get released, the older ones tend to become cheaper over time, so by the time your public beta comes around, there will be better models in scope than there were during prototyping. You'll be glad if your prompt engineering and postprocessing is already optimized for the better models you now can afford.

You probably won't ever want to build and train your own model from scratch, but you may want to take an existing model and *make it your own* by training it specifically on the task that your application will use it for. This process is called *fine-tuning*. While this topic moves beyond the scope of this book, we do want to familiarize you enough with the basic concepts so you'll be able to judge whether it's a promising idea in your case and whether to invest more time in it.

When an LLM is first trained, it effectively reads through lots of documents and learns how to mimic them. In fine-tuning, you present the model with new documents and train it to mimic those documents. This will often decrease the model's ability to produce generic documents, but it can dramatically improve the model's ability to produce the types of documents you anticipate seeing in your work.

To fine-tune a model, you'll need a set of training documents that show successful interactions. These should have factually correct answers, use only the background information you want the model to learn, and adhere to the expected format. How can you gather these examples? You can create some yourself, hire contractors, or even synthesize them. If your app has users, you might collect examples based on success indicators like accepted suggestions or user likes. If your app automates a task previously done by humans, you could use their interactions as examples. Whether you can gather these examples is key in deciding if fine-tuning is worth it (see [Figure 7-8](#)).

Some fine-tuning frameworks allow you to only train your model, surgically, on the portion of the document that addresses the problem, rather than, for example, on the portion of the document where a user specifies the problem. Only focusing on these critical parts of the document is called *loss masking*, and it is useful because probably, you're not interested in whether the model can produce the part of the document that is the prompt.

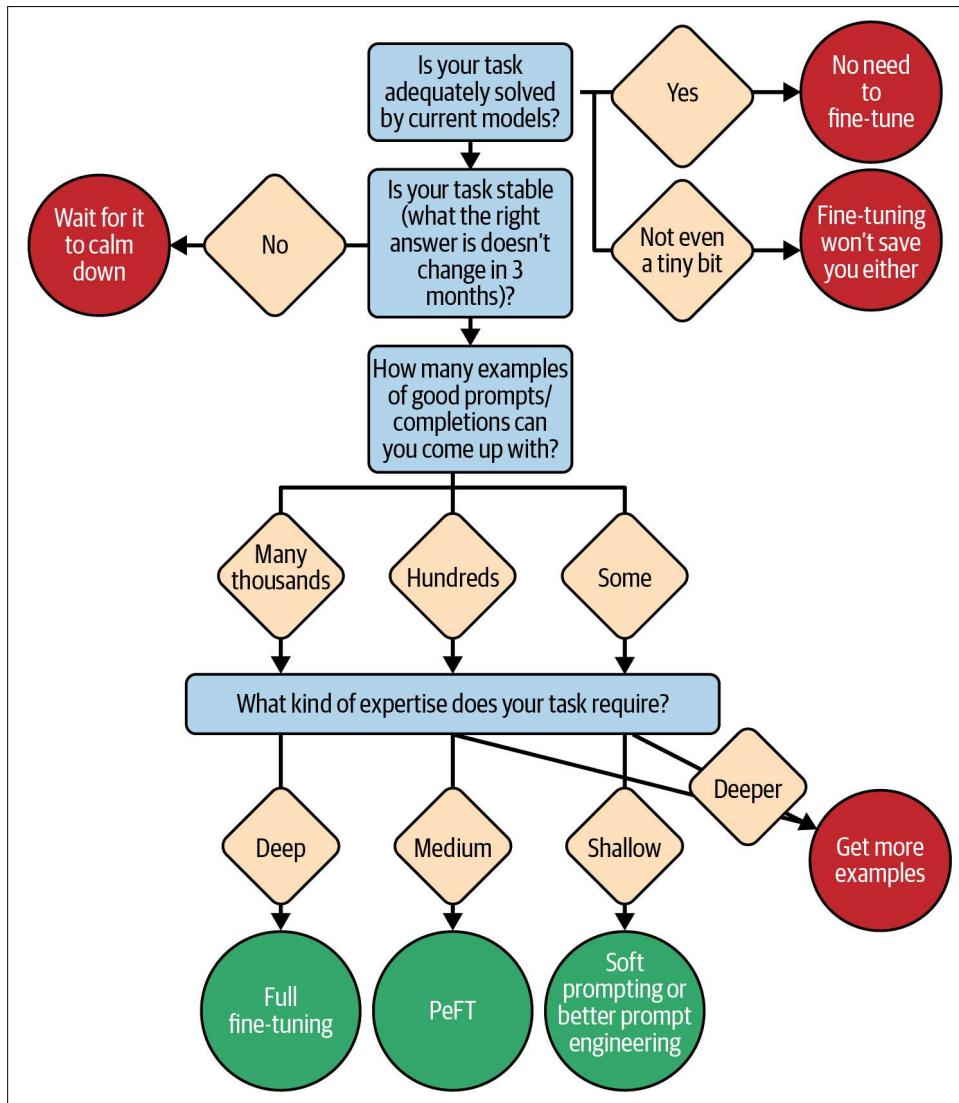


Figure 7-8. Should you fine-tune?

Depending on how many training documents you come up with, you'll have options for different kinds of fine-tuning. We'll discuss the main options here, and we've also summarized them in [Table 7-2](#).

Full fine-tuning, or *continued pre-training*, is simply the continuation of the training process with different documents. That means that every one of the model's billions of parameters is adjusted, and it takes time, computational power, and many, many examples to adjust the parameters in the right way. Like all neural training, this isn't

like explaining a concept to a human and expecting them to learn by understanding. It's more like a riverbed forming: you pour thousands upon thousands of training documents over the model, and very slowly, a groove is carved out. The advantage is, that new groove can be anything. The original model is the starting point, but you can teach it completely new facts and new domains.

Low-rank adaptation (LoRA) is a parameter-efficient fine-tuning technique designed to make model training more efficient. The key idea is that when you don't need the model to learn something entirely new, you don't have to adjust all of its parameters. Instead, LoRA focuses on a few key parameter matrices in the LLM and trains a “*diff*” to those matrices, which for each original matrix is a difference matrix that is added to the original, but one that has fewer degrees of freedom (hence, it's of *low rank*).

This approach has practical benefits—since the *diffs* are small, they can easily be shared between virtual machines, and one deployment can handle multiple *diffs*, allowing you to use the same machine for different models. More importantly, LoRA fine-tuning is relatively fast, typically taking hours or a few days, making it a compute-efficient option.

But there are also drawbacks: depending on the LoRA dimension (a number measuring the degrees of freedom to the *diff* you train), the model is limited in how much it can learn. In general, a good intuition is that LoRA doesn't really teach a model new tricks. Rather, LoRA teaches the model which of the tricks that it's already capable of performing it should expect to use, and in which way. In particular, this includes things like what to pay attention to in the prompt, how to interpret it, and what is expected from the model in the completion. Format and style are easily learnable with LoRA. Another thing that LoRA is great at doing is giving the model a general feel for the prior distributions it should assume for your domain.

Let's explain the last point through an example. Say your application helps people select travel destinations and all your customers are based in Europe. Being Europeans, their preferred suggestions will have a very different distribution than if they were based in the United States. Napa Valley is far away, and Monaco is just around the corner. Fine-tuning can teach the model that, but to be fair, so could you: you could just add to the prompt that the customer is European, so the model should choose destinations based on that. But what about factors you don't know explicitly? Maybe most users of your app are students, and they're looking for budget destinations. Your app telemetry might show that suggesting Monaco usually gets a thumbs-down but that every time you suggest Prague, the user buys a ticket. If you have such data to feed it, LoRA fine-tuning excels at conditioning the model to such a distribution shift, whether it depends on an aspect you're aware of (preference for budget destinations) or not.

With either continued pre-training or LoRA fine-tuning, you can normally get rid of *all* your static prompt context, the general explanations, and instructions—the model will just bake them into its parameters. You also don't need few-shot prompting anymore: all lessons from those few-shots should already be absorbed into the LoRA model, and more effectively than when presented in the prompt. In this sense, fine-tuning is a continuation of prompt engineering by other means.

A technique called *soft prompting* continues further. Thinking back to [Chapter 2](#), consider what happens to the model as it processes the tokens in a prompt. Effectively, the prompt creates a “state of mind” in the model that conditions what tokens it will predict next. So, you can spend a lot of time crafting the words to elicit the right state of mind...or you can simply give a few dozen examples of desired outputs to the model and use machine learning to find a model state that makes the model most likely produce them. Soft prompting is a cool idea, but you'll need to check whether your model framework gives you this opportunity—many don't.

Table 7-2. Different types of fine-tuning

The model typically learns...		It makes the most sense if your training documents number in the...	Fine-tuning often takes...
Full fine-tuning or continued pre-training	New things about a potentially whole new domain.	Tens of thousands.	Weeks or months.
Parameter efficient fine-tuning (e.g., LoRA)	Prior expectations within an existing domain, interpreting information in a certain way, and obeying a fixed format.	Hundreds or thousands.	Days.
Soft prompting	Whatever information is contained in <i>that</i> prompt.	Hundreds.	Hours.

No matter which fine-tuning paradigm you go for, there will be one crucial impact: the Little Red Riding Hood principle will work differently for fine-tuned models. There are now two kinds of documents, two kinds of paths Little Red could follow: the old path of the model's original training and the new path you have fine-tuned for. The old path might be slightly overgrown, but it's still visible, and you need to beware: if the prompt looks like it might follow that path, so will the model in the completion—in effect, the model will simply **forget its fine-tuning**. So, the modified Little Red Riding Hood principle says these things:

1. Try to make your prompt look like the beginning of one of the documents you fine-tuned for.
2. Be very sure not to have it look like one of the original documents instead.

Conclusion

Taming the model is hard: it sometimes feels like LLMs have a mind of their own, and they don't want to follow the path you laid out for them. But you now have a good understanding of how to lead them along the path that you want—do this by clearly defining the completion you want them to provide and then using the tricks you've learned to guide them toward a completion with the expected format, style, and content.

Most of the time, the text of the completion is the focal point of your work. But in this chapter, you also learned about logprobs and how to use them to glean more information from LLM completions. And if the model still won't do as you say, you have the knowledge at your disposal to decide upon a different model or to even train your model yourself, if that's the right path for you.

This chapter concludes what we consider the core prompt-engineering techniques. With a solid understanding of how LLMs work and how to make them work for you, you can now call yourself a proper prompt engineer! But what kind of prompt engineer would be satisfied with merely learning the basics? In the next chapters, we'll discuss advanced techniques that use LLM—mere document completion models—as the central components of flexible agents and powerful workflow execution systems.

PART III

An Expert of the Craft

