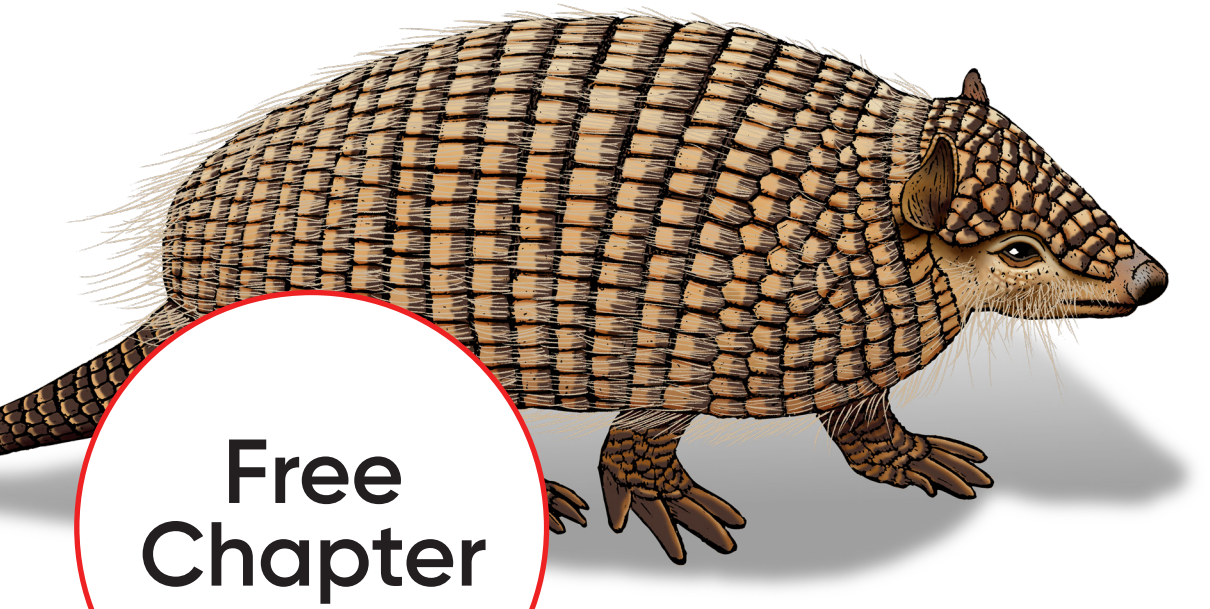
O'REILLY®

Prompt Engineering for Generative AI

Future-Proof Inputs for Reliable AI Outputs



Free Chapter

James Phoenix
& Mike Taylor

Prompt Engineering for Generative AI

Future-Proof Inputs for Reliable AI Outputs at Scale

This excerpt contains Chapter 1. The complete book is available on the O'Reilly Online Learning Platform and through other retailers.

James Phoenix and Mike Taylor

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Prompt Engineering for Generative AI
by James Phoenix and Mike Taylor

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield
Development Editor: Corbin Collins
Copyeditor: Piper Editorial Consulting, LLC
Proofreader: Kim Wimpsett
Indexer: nSight, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

May 2024: First Edition

Revision History for the First Edition
2024-05-15: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098153434* for release details.

978-1-098-15343-4

[LSI]

Table of Contents

1. The Five Principles of Prompting. 1
Overview of the Five Principles of Prompting 4
1. Give Direction 8
2. Specify Format 14
3. Provide Examples 17
4. Evaluate Quality 20
5. Divide Labor 32
Summary 38

CHAPTER 1
The Five Principles of Prompting

Prompt engineering is the process of discovering prompts that reliably yield useful or desired results.

A *prompt* is the input you provide, typically text, when interfacing with an AI model like ChatGPT or Midjourney. The prompt serves as a set of instructions the model uses to predict the desired response: text from *large language models* (LLMs) like ChatGPT, or images from *diffusion models* like Midjourney.

Here is a simple example of a prompt input for a product name generator (inspired by one of OpenAI's examples), and the resulting output from ChatGPT.

Input:

```
Can I have a list of product names for a pair of shoes that
can fit any foot size?
```

Output:

```
Certainly! Here's a list of product names for a pair of shoes
that can fit any foot size:

UniFit SoleStrides
FlexiSize All-Fit Shoes
OmniFit FootFlex
...
AnyStep AdaptiWalkers
FlexiRange UniversalSteps
PerfectGlide MultiFit Shoes
```

This is already a remarkable response for a naive prompt, which feels like magic because we got here with very little effort. As the state-of-the-art models improve, the likelihood you will get *good enough* results on your first try goes up. For any throwaway interactions with an AI, where you don't plan to do the same task again, the naive approach is all you need.

However, if you planned to put this prompt into production, you'd benefit from investing more work into getting it right. Mistakes cost you money in terms of the fees OpenAI charges based on the length of the prompt and response, as well as the time spent fixing mistakes. If you were building a product name generator with thousands of users, there are some obvious issues you'd want attempt to fix:

Vague direction
: You're not briefing the AI on what style of name you want, or what attributes it should have. Do you want a single word or a concatenation? Can the words be made up, or is it important that they're in real English? Do you want the AI to emulate somebody you admire who is famous for great product names?

Unformatted output
: You're getting back a list of separated names line by line, of unspecified length. When you run this prompt multiple times, you'll see sometimes it comes back with a numbered list, and often it has text at the beginning, which makes it hard to parse programmatically.

Missing examples
: You haven't given the AI any examples of what *good* names look like. It's auto-completing using an average of its training data, i.e., the entire internet (with all its inherent bias), but is that what you want? Ideally you'd feed it examples of successful names, common names in an industry, or even just other names you like.

Limited evaluation
: You have no consistent or scalable way to define which names are good or bad, so you have to manually review each response. If you can institute a rating system or other form of measurement, you can optimize the prompt to get better results and identify how many times it fails.

No task division
: You're asking a lot of a single prompt here: there are lots of factors that go into product naming, and this important task is being naively outsourced to the AI all in one go, with no task specialization or visibility into how it's handling this task for you.

Addressing these problems is the basis for the core principles we use throughout this book. There are many different ways to ask an AI model to do the same task, and even slight changes can make a big difference. LLMs work by continuously predicting the next token (approximately three-fourths of a word), starting from what was in your prompt. Each new token is selected based on its probability of appearing next, with an element of randomness (controlled by the *temperature* parameter). As demonstrated in Figure 1-1, the word *shoes* had a lower probability of coming after the start of the name *AnyFit* (0.88%), where a more predictable response would be *Athletic* (72.35%).

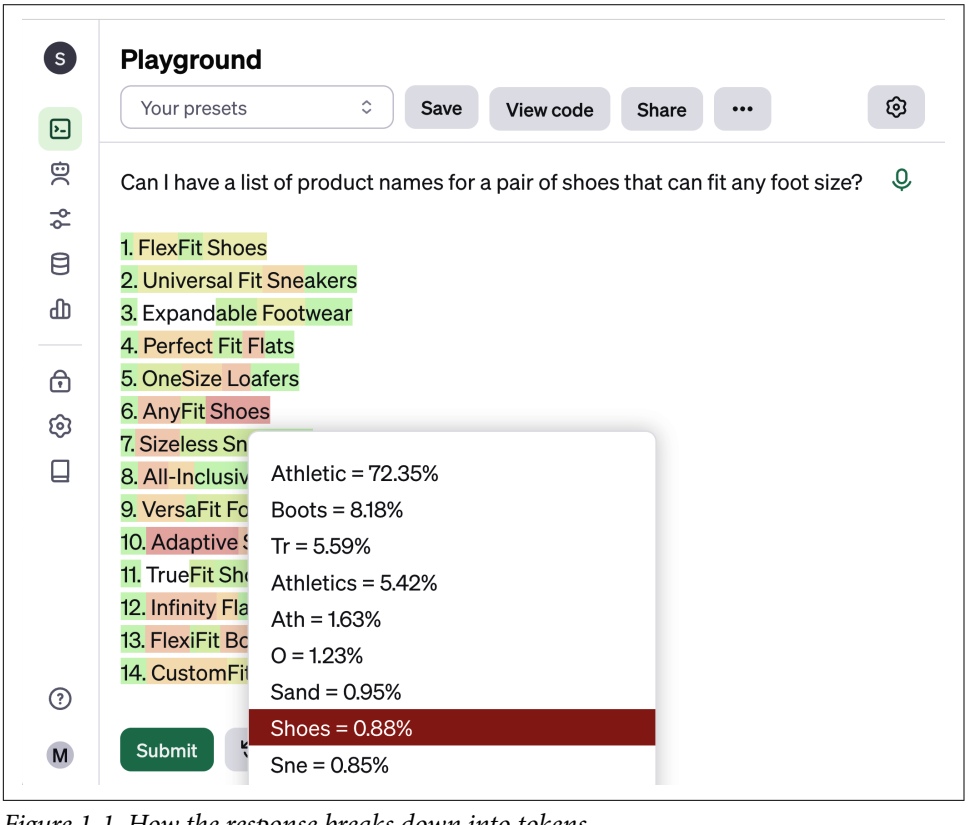


Figure 1-1. How the response breaks down into tokens

LLMs are trained on essentially the entire text of the internet, and are then further fine-tuned to give helpful responses. Average prompts will return average responses, leading some to be underwhelmed when their results don't live up to the hype. What you put in your prompt changes the probability of every word generated, so it matters a great deal to the results you'll get. These models have seen the best and worst of what humans have produced and are capable of emulating almost anything if you know the right way to ask. OpenAI charges based on the number of tokens

used in the prompt and the response, so prompt engineers need to make these tokens count by optimizing prompts for cost, quality, and reliability.

Here's the same example with the application of several prompt engineering techniques. We ask for names in the style of Steve Jobs, state that we want a comma-separated list, and supply examples of the task done well.

Input:

```
Brainstorm a list of product names for a shoe that fits any
foot size, in the style of Steve Jobs.

Return the results as a comma-separated list, in this format:
Product description: A shoe that fits any foot size
Product names: [list of 3 product names]

## Examples
Product description: A refrigerator that dispenses beer
Product names: iBarFridge, iFridgeBeer, iDrinkBeerFridge

Product description: A watch that can tell accurate time in
space
Product names: iNaut, iSpace, iTime

Product description: A home milkshake maker
Product names: iShake, iSmoothie, iShake Mini
```

Output:

```
Product description: A shoe that fits any foot size
Product names: iFitFoot, iPerfectFit, iShoeSize
```

While no prompt is ever perfect, this prompt is optimized to reliably deliver solid product names in the right format. The user of your product name generator can choose somebody other than Steve Jobs to get the types of names they like, they can change the response format if needed, and the output of this prompt can become the input of another. Finally, you could periodically update the examples you use in the prompt based on user feedback, making your system smarter over time.

Overview of the Five Principles of Prompting

The process for optimizing this prompt follows the *Five Principles of Prompting*, which we will dissect using this example in the remainder of this chapter, and recall throughout the book. They map exactly to the five issues we raised when discussing the naive text prompt. You'll find references back to these principles throughout the rest of the book to help you connect the dots to how they're used in practice. The Five Principles of Prompting are as follows:

Give Direction
: Describe the desired style in detail, or reference a relevant persona

Specify Format
: Define what rules to follow, and the required structure of the response

Provide Examples
: Insert a diverse set of test cases where the task was done correctly

Evaluate Quality
: Identify errors and rate responses, testing what drives performance.

Divide Labor
: Split tasks into multiple steps, chained together for complex goals

These principles are not short-lived *tips* or *hacks* but are generally accepted conventions that are useful for working with any level of intelligence, biological or artificial. These principles are model-agnostic and should work to improve your prompt no matter which generative text or image model you're using. We first published these principles in July 2022 in the blog post "Prompt Engineering: From Words to Art and Copy", and they have stood the test of time, including mapping quite closely to OpenAI's own Prompt Engineering Guide, which came a year later. Anyone who works closely with generative AI models is likely to converge on a similar set of strategies for solving common issues, and throughout this book you'll see hundreds of demonstrative examples of how they can be useful for improving your prompts.

We have provided downloadable one-pagers for text and image generation you can use as a checklist when applying these principles. These were created for our popular Udemy course The Complete Prompt Engineering for AI Bootcamp (70,000+ students), which was based on the same principles but with different material to this book.

- Text Generation One-Pager
- Image Generation One-Pager

To show these principles apply equally well to prompting image models, let's use the following example, and explain how to apply each of the Five Principles of Prompting to this specific scenario. Copy and paste the entire input prompt into the Midjourney Bot in Discord, including the link to the image at the beginning, after typing **`/imagine`** to trigger the prompt box to appear (requires a free Discord account, and a paid Midjourney account).

Input:

```
https://s.mj.run/TKAsyhNiKmc stock photo of business meeting
of 4 people watching on white MacBook on top of glass-top
table, Panasonic, DC-GH5
```

Figure 1-2 shows the output.

Figure 1-2. Stock photo of business meeting

This prompt takes advantage of Midjourney's ability to take a base image as an example by uploading the image to Discord and then copy and pasting the URL into the prompt (*https://s.mj.run/TKAsyhNiKmc*), for which the royalty-free image from Unsplash is used (Figure 1-3). If you run into an error with the prompt, try uploading the image yourself and reviewing Midjourney's documentation for any formatting changes.

Figure 1-3. Photo by Mimi Thian on Unsplash

Let's compare this well-engineered prompt to what you get back from Midjourney if you naively ask for a stock photo in the simplest way possible. Figure 1-4 shows an example of what you get without prompt engineering, an image with a darker, more stylistic take on a stock photo than you'd typically expect.

Input:

```
people in a business meeting
```

Figure 1-4 shows the output.

Although less prominent an issue in v5 of Midjourney onwards, community feedback mechanisms (when users select an image to resize to a higher resolution, that choice may be used to train the model) have reportedly biased the model toward a *fantasy* aesthetic, which is less suitable for the stock photo use case. The early adopters of Midjourney came from the digital art world and naturally gravitated toward fantasy and sci-fi styles, which can be reflected in the results from the model even when this aesthetic is not suitable.

Figure 1-4. People in a business meeting

Throughout this book the examples used will be compatiable with ChatGPT Plus (GPT-4) as the text model and Midjourney v6 or Stable Diffusion XL as the image model, though we will specify if it's important. These foundational models are the current state of the art and are good at a diverse range of tasks. The principles are intended to be future-proof as much as is possible, so if you're reading this book when GPT-5, Midjourney v7, or Stable Diffusion XXL is out, or if you're using another vendor like Google, everything you learn here should still prove useful.

1. Give Direction

One of the issues with the naive text prompt discussed earlier was that it wasn't briefing the AI on what *types* of product names you wanted. To some extent, naming a product is a subjective endeavor, and without giving the AI an idea of what names you like, it has a low probability of guessing right.

By the way, a human would also struggle to complete this task without a good *brief*, which is why creative and branding agencies require a detailed briefing on any task from their clients.

Although it's not a perfect mapping, it can be helpful to imagine what context a human might need for this task and try including it in the prompt.

In the example prompt we gave direction through the use of *role-playing*, in that case emulating the style of Steve Jobs, who was famous for iconically naming products. If you change this aspect of the prompt to someone else who is famous in the training data (as well as matching the examples to the right style), you'll get dramatically different results.

Input:

```
Brainstorm a list of product names for a shoe that fits any
foot size, in the style of Elon Musk.

Return the results as a comma-separated list, in this format:
Product description: A shoe that fits any foot size
Product names: [list of 3 product names]

## Examples
Product description: A refrigerator that dispenses beer
Product names: Teslacool, FridgeX, Neuradrink

Product description: A watch that can tell accurate time in
space
Product names: WatchX, iSpace, The Watch Company

Product description: A home milkshake maker
Product names: ShakeX, The Milkshake Company, HyperShake
```

Output:

```
ShoeX, MarsFit, The Shoe Company
```

There are also some rules or best practices you would do well to follow, which could be included in the prompt as context to guide the AI toward a name that works. This technique is sometimes referred to as *prewarming* or *internal retrieval*, and it is simple but effective (Liu et al., 2021). Starting the conversation asking for best practice advice, then asking it to follow its own advice, can help a lot. In effect, you are using it to generate its own direction.

Input:

```
Please give me 5 tips for naming products based on expert
industry advice.
```

Output:

```
Certainly! Naming a product is crucial as it often gives the
first impression to potential customers. Here are five expert
tips to guide you in naming products:

1. **Simplicity & Memorability**: Opt for names that are
short, easy to spell, and memorable. Long, complicated names
can be difficult for customers to recall. Remember, the
easier the name is to remember, the more likely it is that
consumers will think of your product when making a purchase
decision.

...

5. **Positive Connotations**: Names that evoke positive
emotions or imagery can create a favorable perception of
your product. Avoid names that have negative or
controversial associations in any of the primary languages
or cultures where you intend to do business. It's often
worthwhile to conduct linguistic checks and market testing
to ensure there aren't unintended meanings.

Lastly, once you've shortlisted a few names, it's a good
idea to run them by diverse focus groups or conduct surveys
to gauge reactions. Also, ensure that the name you choose is
available for trademarking and doesn't infringe on existing
trademarks.
```

Then within the same chat window, where the model has the context of the past advice it gave, you ask your initial prompt for the task you wanted to complete.

Input:

```
Using this advice, can I have a list of product names for a
pair of shoes that can fit any foot size?
```

Output:

```
Certainly! Based on the criteria given:

1. **Simplicity & Memorability**
2. **Relevance & Descriptiveness**
3. **Differentiation & Uniqueness**
4. **Adaptable & Scalable**
5. **Positive Connotations**

Here's a list of potential product names for shoes that can
fit any foot size:
```

```
1. **FitAll Walkers**: A simple and memorable name that
instantly suggests a universal fit.

...

10. **TrueFit Universals**: Emphasizing genuine comfort and
fit for all.

Before finalizing any name, remember to check for trademarks
to ensure there's no conflict with existing products or
brands.
```

Another fruitful strategy in our experience is to take the best advice out there for the task you want to accomplish and insert that context into the prompt. For example, you could take Brandwatch's 5 Golden Rules for naming a product or another trusted external resource you find, and insert that as context into the prompt. This will increase the length of the prompt significantly, which costs more money (when using the API as a developer), but may be worth the trade-off if the quality of the response improves.

Input:

```
Can I have a list of product names for a pair of shoes that
can fit any foot size? Please follow the five rules below:

1. It should be readable and writable.
If your product name is hard to pronounce, people won't talk about it,
and if they can't write it down (and spell it correctly!)
when they hear it, how do you expect them to Google it?

Keep it simple and don't go with any wacky spellings just for the sake of it.

...

5. It should evoke an emotion, feeling, or idea.
Your product name should tie back into what your product is, what the
feeling you want people to have when experiencing your product is, and/or
what idea are you trying to get across.

It should be emotive and inspiring.
```

Output:

```
1. FlexFit Footwear
2. OmniSize Sneakers
3. AdaptStride Shoes
...
18. OmniComfort Kicks
19. FlexSize Footwear
20. Boundless Soles
```

There are other myriad ways of providing direction. In the image generation example, direction was given by specifying that the business meeting is taking place around a glass-top table. If you change only that detail, you can get a completely different image, as detailed in Figure 1-5.

Input:

```
https://s.mj.run/TKAsyhNiKmc stock photo of business meeting
of four people gathered around a campfire outdoors in the
woods, Panasonic, DC-GH5
```

Figure 1-5 shows the output.

Figure 1-5. Stock photo of business meeting in the woods

Role-playing is also important for image generation, and one of the quite powerful ways you can give Midjourney direction is to supply the name of an artist or art style to emulate. One artist that features heavily in the AI art world is Van Gogh, known for his bold, dramatic brush strokes and vivid use of colors. Watch what happens when you include his name in the prompt, as shown in Figure 1-6.

Input:

```
people in a business meeting, by Van Gogh
```

Figure 1-6 shows the output.

Figure 1-6. People in a business meeting, by Van Gogh

To get that last prompt to work, you need to strip back a lot of the other direction. For example, losing the base image and the words *stock photo* as well as the camera *Panasonic, DC-GH5* helps bring in Van Gogh's style. The problem you may run into is that often with too much direction, the model can quickly get to a conflicting combination that it can't resolve. If your prompt is overly specific, there might not be enough samples in the training data to generate an image that's consistent with all of your criteria. In cases like these, you should choose which element is more important (in this case, Van Gogh) and defer to that.

Direction is one of the most commonly used and broadest principles. It can take the form of simply using the right descriptive words to clarify your intent, or channeling the personas of relevant business celebrities. While too much direction can narrow the creativity of the model, too little direction is the more common problem.

2. Specify Format

AI models are universal translators. Not only does that mean translating from French to English, or Urdu to Klingon, but also between data structures like JSON to YAML, or natural language to Python code. These models are capable of returning a response in almost any format, so an important part of prompt engineering is finding ways to specify what format you want the response to be in.

Every now and again you'll find that the same prompt will return a different format, for example, a numbered list instead of comma separated. This isn't a big deal most of the time, because most prompts are one-offs and typed into ChatGPT or Midjourney. However, when you're incorporating AI tools into production software, occasional flips in format can cause all kinds of errors.

Just like when working with a human, you can avoid wasted effort by specifying up front the format you expect the response to be in. For text generation models, it can often be helpful to output JSON instead of a simple ordered list because that's the universal format for API responses, which can make it simpler to parse and spot errors, as well as to use to render the front-end HTML of an application. YAML is also another popular choice because it enforces a parseable structure while still being simple and human-readable.

In the original prompt you gave direction through both the examples provided, and the colon at the end of the prompt indicated it should complete the list inline. To swap the format to JSON, you need to update both and leave the JSON uncompleted, so GPT-4 knows to complete it.

Input:

```
Return a comma-separated list of product names in JSON for
"A pair of shoes that can fit any foot size.".
Return only JSON.

Examples:
[{
                "Product description": "A home milkshake maker.",
                "Product names": ["HomeShaker", "Fit Shaker",
                "QuickShake", "Shake Maker"]
        },
        {
                "Product description": "A watch that can tell
                accurate time in space.",
                "Product names": ["AstroTime", "SpaceGuard",
                "Orbit-Accurate", "EliptoTime"]}
]
```

Output:

```
[
        {
                "Product description": "A pair of shoes that can \
                fit any foot size.",
                "Product names": ["FlexFit Footwear", "OneSize Step",
                "Adapt-a-Shoe", "Universal Walker"]
        }
]
```

The output we get back is the completed JSON containing the product names. This can then be parsed and used programmatically, in an application or local script. It's also easy from this point to check if there's an error in the formatting using a JSON parser like Python's standard *json* library, because broken JSON will result in a parsing error, which can act as a trigger to retry the prompt or investigate before continuing. If you're still not getting the right format back, it can help to specify at the beginning or end of the prompt, or in the system message if using a chat model: `You are a helpful assistant that only responds in JSON`, or specify JSON output in the model parameters where available (this is called *grammars* with Llama models.

To get up to speed on JSON if you're unfamiliar, W3Schools has a good introduction.

For image generation models, format is very important, because the opportunities for modifying an image are near endless. They range from obvious formats like `stock photo`, `illustration`, and `oil painting`, to more unusual formats like `dashcam footage`, `ice sculpture`, or `in Minecraft` (see Figure 1-7).

Input:

```
business meeting of four people watching on MacBook on top of
table, in Minecraft
```

Figure 1-7 shows the output.

Figure 1-7. Business meeting in Minecraft

When setting a format, it is often necessary to remove other aspects of the prompt that might clash with the specified format. For example, if you supply a base image of a stock photo, the result is some combination of stock photo and the format you wanted. To some degree, image generation models can generalize to new scenarios and combinations they haven't seen before in their training set, but in our experience, the more layers of unrelated elements, the more likely you are to get an unsuitable image.

There is often some overlap between the first and second principles, Give Direction and Specify Format. The latter is about defining what type of output you want, for example JSON format, or the format of a stock photo. The former is about the style of response you want, independent from the format, for example product names in the style of Steve Jobs, or an image of a business meeting in the style of Van Gogh. When there are clashes between style and format, it's often best to resolve them by dropping whichever element is less important to your final result.

3. Provide Examples

The original prompt didn't give the AI any examples of what you think *good* names look like. Therefore, the response is approximate to an average of the internet, and you can do better than that. Researchers would call a prompt with no examples *zero-shot*, and it's always a pleasant surprise when AI can even do a task zero shot: it's a sign of a powerful model. If you're providing zero examples, you're asking for a lot without giving much in return. Even providing one example (*one-shot*) helps considerably, and it's the norm among researchers to test how models perform with multiple examples (*few-shot*). One such piece of research is the famous GPT-3 paper "Language Models are Few-Shot Learners", the results of which are illustrated in Figure 1-8, showing adding one example along with a prompt can improve accuracy in some tasks from 10% to near 50%!

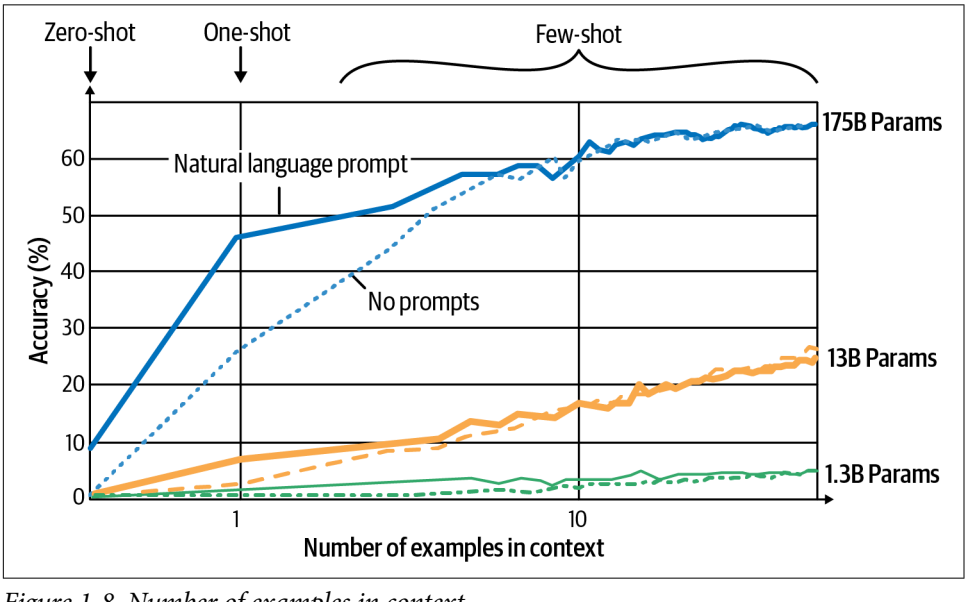


Figure 1-8. Number of examples in context

When briefing a colleague or training a junior employee on a new task, it's only natural that you'd include examples of times that task had previously been done well. Working with AI is the same, and the strength of a prompt often comes down to the examples used. Providing examples can sometimes be easier than trying to explain exactly what it is about those examples you like, so this technique is most effective when you are not a domain expert in the subject area of the task you are attempting to complete. The amount of text you can fit in a prompt is limited (at the time of writing around 6,000 characters on Midjourney and approximately 32,000 characters for the free version of ChatGPT), so a lot of the work of prompt engineering involves selecting and inserting diverse and instructive examples.

There's a trade-off between reliability and creativity: go past three to five examples and your results will become more reliable, while sacrificing creativity. The more examples you provide, and the lesser the diversity between them, the more constrained the response will be to match your examples. If you change all of the examples to animal names in the previous prompt, you'll have a strong effect on the response, which will reliably return only names including animals.

Input:

```
Brainstorm a list of product names for a shoe that fits any
foot size.

Return the results as a comma-separated list, in this format:
Product description: A shoe that fits any foot size
Product names: [list of 3 product names]

## Examples:
Product description: A home milkshake maker.
Product names: Fast Panda, Healthy Bear, Compact Koala

Product description: A watch that can tell accurate time in
space.
Product names: AstroLamb, Space Bear, Eagle Orbit

Product description: A refrigerator that dispenses beer
Product names: BearFridge, Cool Cat, PenguinBox
```

Output:

```
Product description: A shoe that fits any foot size
Product names: FlexiFox, ChameleonStep, PandaPaws
```

Of course this runs the risk of missing out on returning a much better name that doesn't fit the limited space left for the AI to play in. Lack of diversity and variation in examples is also a problem in handling edge cases, or uncommon scenarios. Including one to three examples is easy and almost always has a positive effect, but above that number it becomes essential to experiment with the number of examples you include, as well as the similarity between them. There is some evidence (Hsieh et al., 2023) that direction works better than providing examples, and it typically isn't straightforward to collect good examples, so it's usually prudent to attempt the principle of Give Direction first.

In the image generation space, providing examples usually comes in the form of providing a base image in the prompt, called *img2img* in the open source Stable Diffusion community. Depending on the image generation model being used, these images can be used as a starting point for the model to generate from, which greatly affects the results. You can keep everything about the prompt the same but swap out the provided base image for a radically different effect, as in Figure 1-9.

Input:

```
stock photo of business meeting of 4 people watching on
white MacBook on top of glass-top table, Panasonic, DC-GH5
```

Figure 1-9 shows the output.

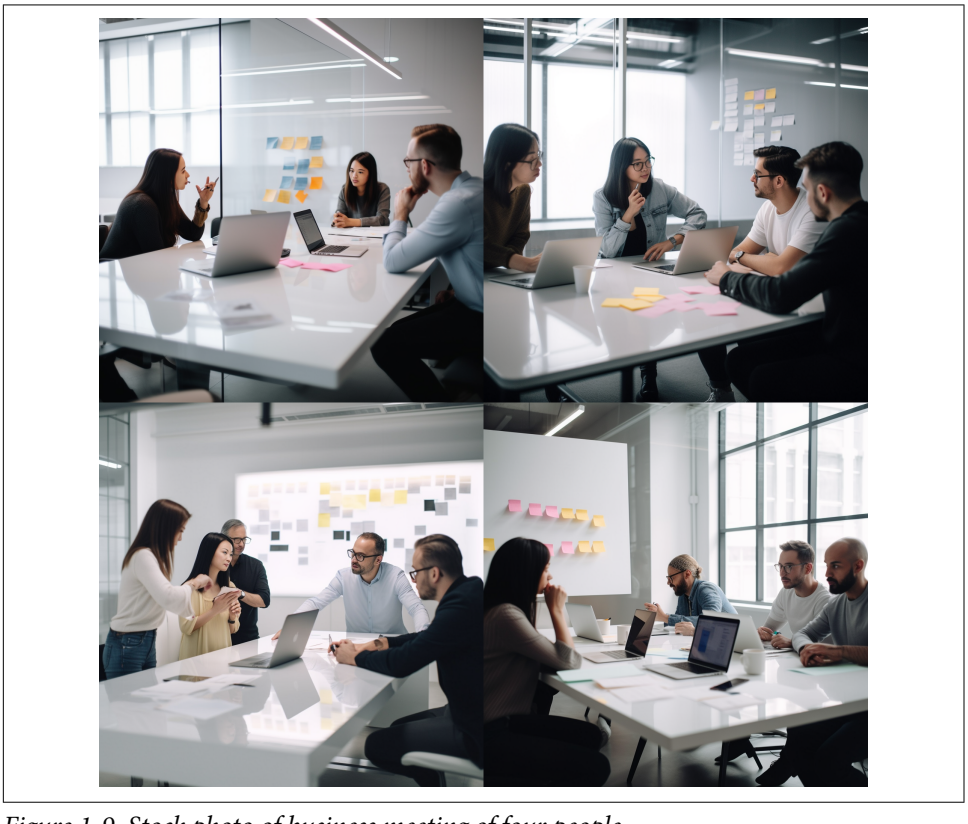


Figure 1-9. Stock photo of business meeting of four people

In this case, by substituting for the image shown in Figure 1-10, also from Unsplash, you can see how the model was pulled in a different direction and incorporates whiteboards and sticky notes now.

These examples demonstrate the capabilities of image generation models, but we would exercise caution when uploading base images for use in prompts. Check the licensing of the image you plan to upload and use in your prompt as the base image, and avoid using clearly copyrighted images. Doing so can land you in legal trouble and is against the terms of service for all the major image generation model providers.

Figure 1-10. Photo by Jason Goodman on Unsplash

4. Evaluate Quality

As of yet, there has been no feedback loop to judge the quality of your responses, other than the basic trial and error of running the prompt and seeing the results, referred to as *blind prompting*. This is fine when your prompts are used temporarily for a single task and rarely revisited. However, when you're reusing the same prompt multiple times or building a production application that relies on a prompt, you need to be more rigorous with measuring results.

There are a number of ways performance can be evaluated, and it depends largely on what tasks you're hoping to accomplish. When a new AI model is released, the focus tends to be on how well the model did on *evals* (evaluations), a standardized set of questions with predefined answers or grading criteria that are used to test performance across models. Different models perform differently across different types of tasks, and there is no guarantee a prompt that worked previously will translate well to a new model. OpenAI has made its evals framework for benchmarking performance of LLMs open source and encourages others to contribute additional eval templates.

In addition to the standard academic evals, there are also more headline-worthy tests like GPT-4 passing the bar exam. Evaluation is difficult for more subjective tasks, and can be time-consuming or prohibitively costly for smaller teams. In some instances researchers have turned to using more advanced models like GPT-4 to evaluate responses from less sophisticated models, as was done with the release of Vicuna-13B, a fine-tuned model based on Meta's Llama open source model (see Figure 1-11).

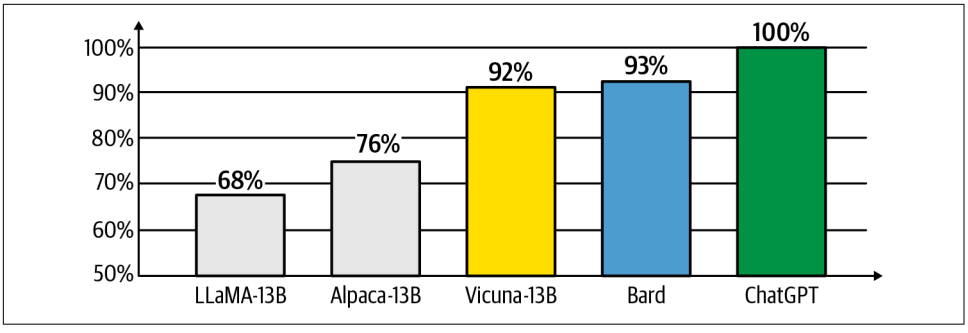


Figure 1-11. Vicuna GPT-4 Evals

More rigorous evaluation techniques are necessary when writing scientific papers or grading a new foundation model release, but often you will only need to go just one step above basic trial and error. You may find that a simple thumbs-up/thumbs-down rating system implemented in a Jupyter Notebook can be enough to add some rigor to prompt optimization, without adding too much overhead. One common test is to see whether providing examples is worth the additional cost in terms of prompt length, or whether you can get away with providing no examples in the prompt. The first step is getting responses for multiple runs of each prompt and storing them in a spreadsheet, which we will do after setting up our environment.

You can install the OpenAI Python package with `pip install openai`. If you're running into compatability issues with this package, create a virtual environment and install our *requirements.txt* (instructions in the preface).

To utilize the API, you'll need to create an OpenAI account and then navigate here for your API key.

Hardcoding API keys in scripts is not recommended due to security reasons. Instead, utilize environment variables or configuration files to manage your keys.

Once you have an API key, it's crucial to assign it as an environment variable by executing the following command, replacing `api_key` with your actual API key value:

```
export OPENAI_API_KEY="api_key"
```

Or on Windows:

```
set OPENAI_API_KEY=api_key
```

Alternatively, if you'd prefer not to preset an API key, then you can manually set the key while initializing the model, or load it from an *.env* file using *python-dotenv*. First, install the library with `pip install python-dotenv`, and then load the environment variables with the following code at the top of your script or notebook:

```
from dotenv import load_dotenv

load_dotenv()  # take environment variables from .env.
```

The first step is getting responses for multiple runs of each prompt and storing them in a spreadsheet.

Input:

```
# Define two variants of the prompt to test zero-shot
# vs few-shot
prompt_A = """Product description: A pair of shoes that can
fit any foot size.
Seed words: adaptable, fit, omni-fit.
Product names:"""

prompt_B = """Product description: A home milkshake maker.
Seed words: fast, healthy, compact.
Product names: HomeShaker, Fit Shaker, QuickShake, Shake
Maker

Product description: A watch that can tell accurate time in
space.
Seed words: astronaut, space-hardened, eliptical orbit
Product names: AstroTime, SpaceGuard, Orbit-Accurate,
EliptoTime.

Product description: A pair of shoes that can fit any foot
size.
Seed words: adaptable, fit, omni-fit.
Product names:"""

test_prompts = [prompt_A, prompt_B]

import pandas as pd
from openai import OpenAI
import os

# Set your OpenAI key as an environment variable
# https://platform.openai.com/api-keys
client = OpenAI(
  api_key=os.environ['OPENAI_API_KEY'],  # Default
)

def get_response(prompt):
    response = client.chat.completions.create(
        model="gpt-3.5-turbo",
```

```
            messages=[
                {
                    "role": "system",
                    "content": "You are a helpful assistant."
                },
                {
                    "role": "user",
                    "content": prompt
                }
            ]
        )
        return response.choices[0].message.content

    # Iterate through the prompts and get responses
    responses = []
    num_tests = 5

    for idx, prompt in enumerate(test_prompts):
        # prompt number as a letter
        var_name = chr(ord('A') + idx)

        for i in range(num_tests):
            # Get a response from the model
            response = get_response(prompt)

            data = {
                "variant": var_name,
                "prompt": prompt,
                "response": response
                }
            responses.append(data)

    # Convert responses into a dataframe
    df = pd.DataFrame(responses)

    # Save the dataframe as a CSV file
    df.to_csv("responses.csv", index=False)

    print(df)
```

Output:

```
  variant                                             prompt
  \
0       A  Product description: A pair of shoes that can ...
1       A  Product description: A pair of shoes that can ...
2       A  Product description: A pair of shoes that can ...
3       A  Product description: A pair of shoes that can ...
4       A  Product description: A pair of shoes that can ...
5       B  Product description: A home milkshake maker.\n...
6       B  Product description: A home milkshake maker.\n...
7       B  Product description: A home milkshake maker.\n...
8       B  Product description: A home milkshake maker.\n...
```

```
9       B  Product description: A home milkshake maker.\n...

                                              response
0  1. Adapt-a-Fit Shoes \n2. Omni-Fit Footwear \n...
1  1. OmniFit Shoes\n2. Adapt-a-Sneaks \n3. OneFi...
2  1. Adapt-a-fit\n2. Flexi-fit shoes\n3. Omni-fe...
3  1. Adapt-A-Sole\n2. FitFlex\n3. Omni-FitX\n4. ...
4  1. Omni-Fit Shoes\n2. Adapt-a-Fit Shoes\n3. An...
5  Adapt-a-Fit, Perfect Fit Shoes, OmniShoe, OneS...
6       FitAll, OmniFit Shoes, SizeLess, AdaptaShoes
7       AdaptaFit, OmniShoe, PerfectFit, AllSizeFit.
8  FitMaster, AdaptoShoe, OmniFit, AnySize Footwe...
9        Adapt-a-Shoe, PerfectFit, OmniSize, FitForm
```

Here we're using the OpenAI API to generate model responses to a set of prompts and storing the results in a dataframe, which is saved to a CSV file. Here's how it works:

1. Two prompt variants are defined, and each variant consists of a product description, seed words, and potential product names, but `prompt_B` provides two examples.
2. Import statements are called for the Pandas library, OpenAI library, and os library.
3. The `get_response` function takes a prompt as input and returns a response from the `gpt-3.5-turbo` model. The prompt is passed as a user message to the model, along with a system message to set the model's behavior.
4. Two prompt variants are stored in the `test_prompts` list.
5. An empty list `responses` is created to store the generated responses, and the variable `num_tests` is set to 5.
6. A nested loop is used to generate responses. The outer loop iterates over each prompt, and the inner loop generates `num_tests` (five in this case) number of responses per prompt.
   a. The `enumerate` function is used to get the index and value of each prompt in `test_prompts`. This index is then converted to a corresponding uppercase letter (e.g., 0 becomes *A*, 1 becomes *B*) to be used as a variant name.
   b. For each iteration, the `get_response` function is called with the current prompt to generate a response from the model.
   c. A dictionary is created with the variant name, the prompt, and the model's response, and this dictionary is appended to the `responses` list.
7. Once all responses have been generated, the `responses` list (which is now a list of dictionaries) is converted into a Pandas DataFrame.

8. This dataframe is then saved to a CSV file with the Pandas built-in `to_csv` function, making the file *responses.csv* with `index=False` so as to not write row indices.
9. Finally, the dataframe is printed to the console.

Having these responses in a spreadsheet is already useful, because you can see right away even in the printed response that `prompt_A` (zero-shot) in the first five rows is giving us a numbered list, whereas `prompt_B` (few-shot) in the last five rows tends to output the desired format of a comma-separated inline list. The next step is to give a rating on each of the responses, which is best done blind and randomized to avoid favoring one prompt over another.

Input:

```
import ipywidgets as widgets
from IPython.display import display
import pandas as pd

# load the responses.csv file
df = pd.read_csv("responses.csv")

# Shuffle the dataframe
df = df.sample(frac=1).reset_index(drop=True)

# df is your dataframe and 'response' is the column with the
# text you want to test
response_index = 0
# add a new column to store feedback
df['feedback'] = pd.Series(dtype='str')

def on_button_clicked(b):
    global response_index
    #  convert thumbs up / down to 1 / 0
    user_feedback = 1 if b.description == "\U0001F44D" else 0

    # update the feedback column
    df.at[response_index, 'feedback'] = user_feedback

    response_index += 1
    if response_index < len(df):
        update_response()
    else:
        # save the feedback to a CSV file
        df.to_csv("results.csv", index=False)

        print("A/B testing completed. Here's the results:")
        # Calculate score and num rows for each variant
        summary_df = df.groupby('variant').agg(
            count=('feedback', 'count'),
            score=('feedback', 'mean')).reset_index()
```

```
        print(summary_df)

def update_response():
    new_response = df.iloc[response_index]['response']
    if pd.notna(new_response):
        new_response = "<p>" + new_response + "</p>"
    else:
        new_response = "<p>No response</p>"
    response.value = new_response
    count_label.value = f"Response: {response_index + 1}"
    count_label.value += f"/{len(df)}"

response = widgets.HTML()
count_label = widgets.Label()

update_response()

thumbs_up_button = widgets.Button(description='\U0001F44D')
thumbs_up_button.on_click(on_button_clicked)

thumbs_down_button = widgets.Button(
    description='\U0001F44E')
thumbs_down_button.on_click(on_button_clicked)

button_box = widgets.HBox([thumbs_down_button,
thumbs_up_button])

display(response, button_box, count_label)
```

The output is shown in Figure 1-12:

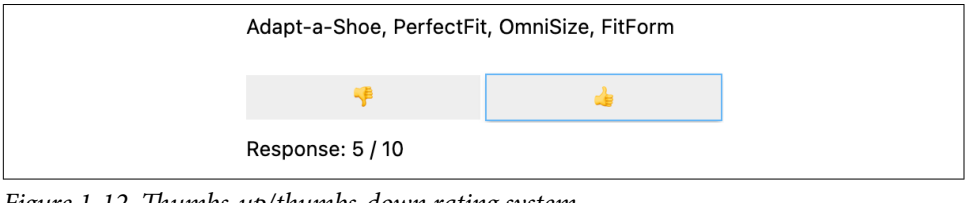


Figure 1-12. Thumbs-up/thumbs-down rating system

If you run this in a Jupyter Notebook, a widget displays each AI response, with a thumbs-up or thumbs-down button (see Figure 1-12) This provides a simple interface for quickly labeling responses, with minimal overhead. If you wish to do this outside of a Jupyter Notebook, you could change the thumbs-up and thumbs-down emojis for *Y* and *N*, and implement a loop using the built-in `input()` function, as a text-only replacement for iPyWidgets.

Once you've finished labeling the responses, you get the output, which shows you how each prompt performs.

Output:

```
A/B testing completed. Here's the results:
  variant  count  score
0       A      5    0.2
1       B      5    0.6
```

The dataframe was shuffled at random, and each response was labeled blind (without seeing the prompt), so you get an accurate picture of how often each prompt performed. Here is the step-by-step explanation:

1. Three modules are imported: `ipywidgets`, `IPython.display`, and `pandas`. `ipywidgets` contains interactive HTML widgets for Jupyter Notebooks and the IPython kernel. `IPython.display` provides classes for displaying various types of output like images, sound, displaying HTML, etc. Pandas is a powerful data manipulation library.
2. The pandas library is used to read in the CSV file *responses.csv*, which contains the responses you want to test. This creates a Pandas DataFrame called `df`.
3. `df` is shuffled using the `sample()` function with `frac=1`, which means it uses all the rows. The `reset_index(drop=True)` is used to reset the indices to the standard 0, 1, 2, …, n index.
4. The script defines `response_index` as 0. This is used to track which response from the dataframe the user is currently viewing.
5. A new column `feedback` is added to the dataframe `df` with the data type as `str` or string.
6. Next, the script defines a function `on_button_clicked(b)`, which will execute whenever one of the two buttons in the interface is clicked.
   a. The function first checks the `description` of the button clicked was the thumbs-up button (`\U0001F44D`; 👍), and sets `user_feedback` as 1, or if it was the thumbs-down button (`\U0001F44E` 👎), it sets `user_feedback` as 0.
   b. Then it updates the `feedback` column of the dataframe at the current `response_index` with `user_feedback`.
   c. After that, it increments `response_index` to move to the next response.
   d. If `response_index` is still less than the total number of responses (i.e., the length of the dataframe), it calls the function `update_response()`.
   e. If there are no more responses, it saves the dataframe to a new CSV file *results.csv*, then prints a message, and also prints a summary of the results by variant, showing the count of feedback received and the average score (mean) for each variant.

7. The function `update_response()` fetches the next response from the dataframe, wraps it in paragraph HTML tags (if it's not null), updates the `response` widget to display the new response, and updates the `count_label` widget to reflect the current response number and total number of responses.
8. Two widgets, `response` (an HTML widget) and `count_label` (a Label widget), are instantiated. The `update_response()` function is then called to initialize these widgets with the first response and the appropriate label.
9. Two more widgets, `thumbs_up_button` and `thumbs_down_button` (both Button widgets), are created with thumbs-up and thumbs-down emoji as their descriptions, respectively. Both buttons are configured to call the `on_button_clicked()` function when clicked.
10. The two buttons are grouped into a horizontal box (`button_box`) using the `HBox` function.
11. Finally, the `response`, `button_box`, and `count_label` widgets are displayed to the user using the `display()` function from the `IPython.display` module.

A simple rating system such as this one can be useful in judging prompt quality and encountering edge cases. Usually in less than 10 test runs of a prompt you uncover a deviation, which you otherwise wouldn't have caught until you started using it in production. The downside is that it can get tedious rating lots of responses manually, and your ratings might not represent the preferences of your intended audience. However, even small numbers of tests can reveal large differences between two prompting strategies and reveal nonobvious issues before reaching production.

Iterating on and testing prompts can lead to radical decreases in the length of the prompt and therefore the cost and latency of your system. If you can find another prompt that performs equally as well (or better) but uses a shorter prompt, you can afford to scale up your operation considerably. Often you'll find in this process that many elements of a complex prompt are completely superfluous, or even counterproductive.

The *thumbs-up* or other manually labeled indicators of quality don't have to be the only judging criteria. Human evaluation is generally considered to be the most accurate form of feedback. However, it can be tedious and costly to rate many samples manually. In many cases, as in math or classification use cases, it may be possible to establish *ground truth* (reference answers to test cases) to programmatically rate the results, allowing you to scale up considerably your testing and monitoring efforts. The following is not an exhaustive list because there are many motivations for evaluating your prompt programmatically:

Cost
: Prompts that use a lot of tokens, or work only with more expensive models, might be impractical for production use.

Latency
: Equally the more tokens there are, or the larger the model required, the longer it takes to complete a task, which can harm user experience.

Calls
: Many AI systems require multiple calls in a loop to complete a task, which can seriously slow down the process.

Performance
: Implement some form of external feedback system, for example a physics engine or other model for predicting real-world results.

Classification
: Determine how often a prompt correctly labels given text, using another AI model or rules-based labeling.

Reasoning
: Work out which instances the AI fails to apply logical reasoning or gets the math wrong versus reference cases.

Hallucinations
: See how frequently you encouner hallucinations, as measured by invention of new terms not included in the prompt's context.

Safety
: Flag any scenarios where the system might return unsafe or undesirable results using a safety filter or detection system.

Refusals
: Find out how often the system incorrectly refuses to fulfill a reasonable user request by flagging known refusal language.

Adversarial
: Make the prompt robust against known prompt injection attacks that can get the model to run undesirable prompts instead of what you programmed.

Similarity
: Use shared words and phrases (BLEU or ROGUE) or vector distance (explained in Chapter 5) to measure similarity between generated and reference text.

Once you start rating which examples were good, you can more easily update the examples used in your prompt as a way to continuously make your system smarter over time. The data from this feedback can also feed into examples for fine-tuning, which starts to beat prompt engineering once you can supply a few thousand examples, as shown in Figure 1-13.

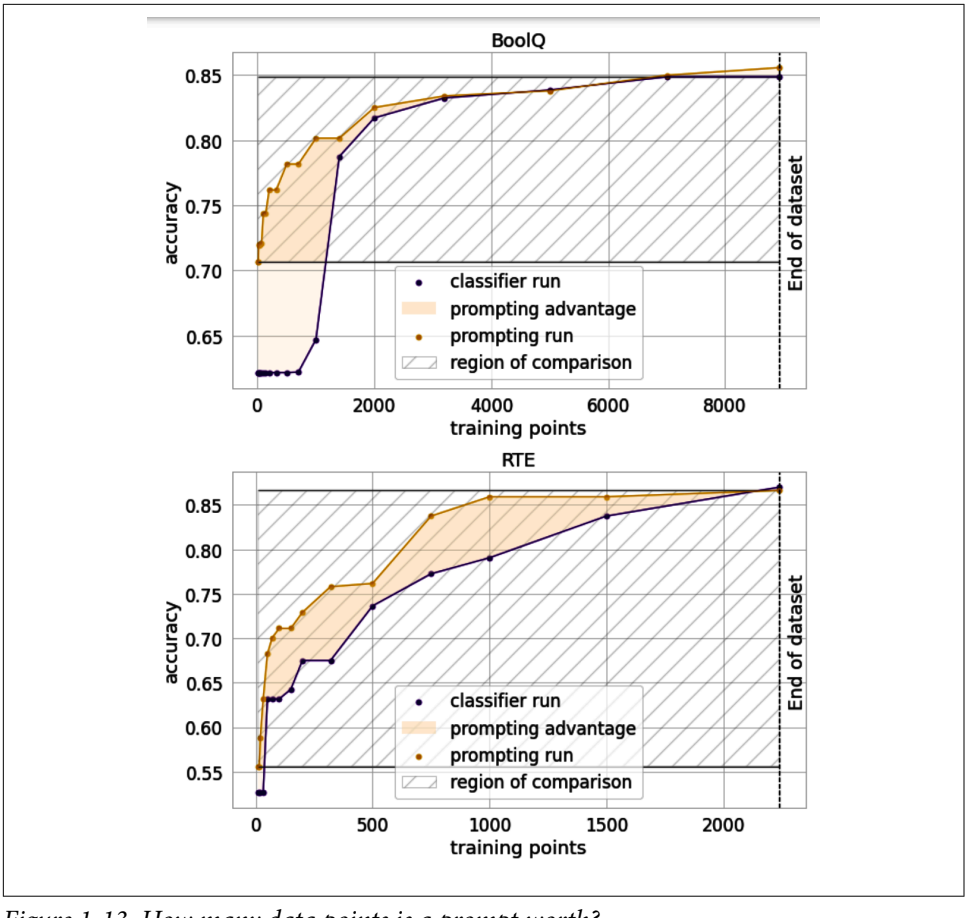


Figure 1-13. How many data points is a prompt worth?

Graduating from thumbs-up or thumbs-down, you can implement a 3-, 5-, or 10-point rating system to get more fine-grained feedback on the quality of your prompts. It's also possible to determine aggregate relative performance through comparing responses side by side, rather than looking at responses one at a time. From this you can construct a fair across-model comparison using an *Elo rating*, as is popular in chess and used in the Chatbot Arena by *lmsys.org*.

For image generation, evaluation usually takes the form of *permutation* prompting, where you input multiple directions or formats and generate an image for each

combination. Images can than be scanned or later arranged in a grid to show the effect that different elements of the prompt can have on the final image.

Input:

```
{stock photo, oil painting, illustration} of business
meeting of {four, eight} people watching on white MacBook on
top of glass-top table
```

In Midjourney this would be compiled into six different prompts, one for every combination of the three formats (stock photo, oil painting, illustration) and two numbers of people (four, eight).

Input:

```
1. stock photo of business meeting of four people watching
on white MacBook on top of glass-top table

2. stock photo of business meeting of eight people watching
on white MacBook on top of glass-top table

3. oil painting of business meeting of four people watching
on white MacBook on top of glass-top table

4. oil painting of business meeting of eight people watching
on white MacBook on top of glass-top table

5. illustration of business meeting of four people watching
on white MacBook on top of glass-top table

6. illustration of business meeting of eight people watching
on white MacBook on top of glass-top table
```

Each prompt generates its own four images as usual, which makes the output a little harder to see. We have selected one from each prompt to upscale and then put them together in a grid, shown as Figure 1-14. You'll notice that the model doesn't always get the correct number of people (generative AI models are surprisingly bad at math), but it has correctly inferred the general intention by adding more people to the photos on the right than the left.

Figure 1-14 shows the output.

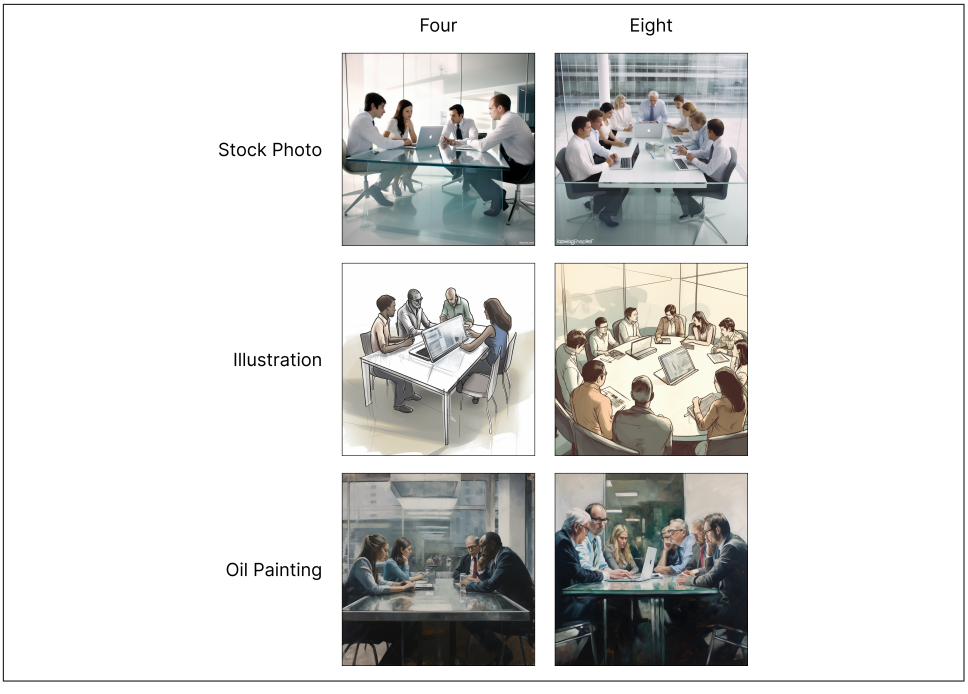


Figure 1-14. Prompt permutations grid

With models that have APIs like Stable Diffusion, you can more easily manipulate the photos and display them in a grid format for easy scanning. You can also manipulate the random seed of the image to fix a style in place for maximum reproducibility. With image classifiers it may also be possible to programmatically rate images based on their safe content, or if they contain certain elements associated with success or failure.

5. Divide Labor

As you build out your prompt, you start to get to the point where you're asking a lot in a single call to the AI. When prompts get longer and more convoluted, you may find the responses get less deterministic, and hallucinations or anomalies increase. Even if you manage to arrive at a reliable prompt for your task, that task is likely just one of a number of interrelated tasks you need to do your job. It's natural to start exploring how many other of these tasks could be done by AI and how you might string them together.

One of the core principles of engineering is to use task decomposition to break problems down into their component parts, so you can more easily solve each individual problem and then reaggregate the results. Breaking your AI work into multiple calls that are chained together can help you accomplish more complex tasks, as well as provide more visibility into what part of the chain is failing.

There are lots of factors that go into product naming, and an important task is naively outsourced to the AI with no visibility into how it's weighing the importance of these factors (if at all). The way our current system works, we're getting a list of names, but all names are displayed with equal importance with no further context for helping us decide. Fortunately AI tools are capable of self-evaluation; if we add a second step to our task, we can automatically check for nondesirable outputs.

Input:

```
Rate this list of product names for a pair of shoes that can
fit any foot size. The rating should be out of 10, inline
next to the product name:

UniFit SoleStrides
FlexiSize All-Fit Shoes
OmniFit FootFlex
...
AnyStep AdaptiWalkers
FlexiRange UniversalSteps
PerfectGlide MultiFit Shoes
```

Output:

```
UniFit SoleStrides: 7/10
FlexiSize All-Fit Shoes: 8/10
OmniFit FootFlex: 7/10
...
AnyStep AdaptiWalkers: 7/10
FlexiRange UniversalSteps: 7.5/10
PerfectGlide MultiFit Shoes: 9/10

This ranking is based on clarity, memorability, and how well
the name communicates the unique selling point of the
product (that they can fit any foot size). Remember, these
ratings are subjective and the ultimate success of a product
name also depends on the target market, branding, and
marketing efforts.
```

In running this multiple times, it consistently rates the name "OneSize Glovewalkers" as the worst, providing context (if you ask) that the concept might be confusing in a shoe context. You may be wondering why, if the model *knows* this is a bad name, does it suggest it in the first place? LLMs work by predicting the next token in a sequence and therefore struggle to know what the overall response will be when

finished. However, when it has all the tokens from a previous response to review, it can more easily predict whether this would be labeled as a good or bad response.

We can continue to chain multiple calls together to improve the results of our task. For example, we could split this into three separate ratings: clarity, memorability, and how well the name communicates the unique selling point of the product. These ratings could then be given to a human as additional context on the final decision, or even calculated together to select the final name programmatically.

The real unlock in learning to work professionally with AI versus just playing around with prompting is realizing that every part of the system can be broken down into a series of iterative steps. Even with a single prompt this principles applies, as simply appending `Let's think step by step` to the prompt can lead to demonstrable gains in reasoning and proficiency, as well as provide an audit trail for quality assurance and debugging. When taking the time and tokens to reason, the ratings change and are more consistent with the scoring criteria.

Input:

```
Let's think step by step. Rate this list of product names
for a pair of shoes that can fit any foot size. The rating
should be out of 10, inline next to the product name:

UniFit SoleStrides
FlexiSize All-Fit Shoes
OmniFit FootFlex
...
AnyStep AdaptiWalkers
FlexiRange UniversalSteps
PerfectGlide MultiFit Shoes
```

Output:

```
Here's a rating out of 10 for each product name, considering
factors like descriptiveness, catchiness, and overall
appeal:

1. **UniFit SoleStrides**: 7/10 - Sounds unifying but may
not immediately convey the "fit any size" idea.
2. **FlexiSize All-Fit Shoes**: 8.5/10 - Combines
flexibility and the notion that it fits all sizes.
3. **OmniFit FootFlex**: 8/10 - Omnipresence (fitting all)
and flexibility nicely conveyed.
...
18. **AnyStep AdaptiWalkers**: 7.5/10 - "AnyStep" suggests
versatility, but the overall name might be less direct.
19. **FlexiRange UniversalSteps**: 8/10 - Combines
flexibility and universal fit.
20. **PerfectGlide MultiFit Shoes**: 8.5/10 - "MultiFit" is
clear, and "PerfectGlide" suggests a comfortable experience.
```

```
Remember, the best name can also be influenced by your
target audience, branding strategy, and the overall message
you want to convey.
```

OpenAI calls this "giving the model time to think," and it is a key tenet of prompt engineering. In effect, *chain of thought* techniques like this, where the model is encouraged to list out its steps, are like dividing a task within the same prompt. Once we've automated product naming given a product idea, we can call ChatGPT again to describe each product, which in turn can be fed into Midjourney to generate an image of each product. Using an AI model to generate a prompt for an AI model is *meta prompting*, and it works because LLMs are human-level prompt engineers (Zhou, 2022).

Input:

```
Describe the new product "OneFit UltraStride Shoes," a shoe
that can fit any foot size, in great detail:
```

Output:

```
Introducing the revolutionary "OneFit UltraStride Shoes,"
the footwear innovation designed to fit any foot size with
unparalleled comfort and style. These shoes are a result of
cutting-edge technology and meticulous craftsmanship,
ensuring a truly personalized and enjoyable walking or
running experience for everyone.

Design and Materials:
The OneFit UltraStride Shoes boast an elegant and modern
design that appeals to both fashion-forward individuals and
sports enthusiasts. They feature a sleek and streamlined
silhouette, crafted using premium materials for durability
and aesthetic appeal. The shoe's upper is constructed from a
combination of breathable mesh and flexible synthetic
materials, allowing for excellent airflow and a lightweight
feel.

...

Color and Customization:
OneFit UltraStride Shoes are available in a variety of
vibrant colors and stylish patterns, catering to diverse
personal preferences. You can choose from classic
monochromatic options for a timeless look or opt for bold
and vibrant hues to make a fashion statement. The ability to
customize the shoe's appearance allows individuals to
express their unique style while enjoying the benefits of
the OneFit technology.

Conclusion:
The OneFit UltraStride Shoes represent a breakthrough in
footwear technology, eliminating the hassle of finding the
```

```
perfect fit for your feet. With their adaptive design,
superior comfort, and stylish aesthetics, these shoes offer
a truly personalized experience for individuals of all foot
sizes. Stay comfortable, confident, and on-trend with the
OneFit UltraStride Shoes, the ultimate choice for those
seeking unmatched versatility and a perfect fit.
```

DALL-E is well-known by GPT-4, and therefore you can invoke its name within ChatGPT and it does a reasonable job at crafting a prompt for an image generation tool. If you were planning on using this prompt in production, you may consider applying the prompting principle of providing examples, but it does a good enough job for our purposes without examples.

Input:

```
Convert this description into a short prompt for an AI image
generation model such as DALL-E:
```

Output:

```
Create an eye-catching image that visually represents the
"OneFit UltraStride Shoes," a revolutionary footwear product
that adapts to any foot size. Showcase a sleek and modern
shoe design incorporating a stretchable upper, an adjustable
lacing system, and a cushioned insole for optimal comfort.
Emphasize the shoe's versatility by displaying it in action
on various surfaces, highlighting its excellent traction.
Incorporate vibrant colors and patterns to convey the
customizable nature of the product. The image should capture
the essence of innovation, style, and personalized fit that
the OneFit UltraStride Shoes offer.
```

The output of this prompt can now be plugged into image generation tools like DALL-E or Midjourney as a prompt, which can give you a good starting point for visualizing what the product might look like. Although this might not be the final design you go with, seeing an image is more evocative and helps people form an opinion faster. It's easier cognitively to criticize or compliment an existing image than it is to imagine a new image from a blank page or section of text.

Figure 1-15 shows the output.

Figure 1-15. OneFit UltraStride shoes

It's common practice when working with AI professionally to chain multiple calls to AI together, and even multiple models, to accomplish more complex goals. Even single-prompt applications are often built dynamically, based on outside context queried from various databases or other calls to an AI model. The library LangChain has developed tooling for chaining multiple prompt templates and queries together, making this process more observable and well structured. A foundational example is progressive summarization, where text that is too large to fit into a context window can be split into multiple chunks of text, with each being summarized, before finally summarizing the summaries. If you talk to builders of early AI products, you'll find they're all under the hood chaining multiple prompts together, called *AI chaining*, to accomplish better results in the final output.

The Reason and Act (ReAct) framework was one of the first popular attempts at AI agents, including the open source projects BabyAGI, AgentGPT and Microsoft AutoGen. In effect, these agents are the result of chaining multiple AI calls together in order to plan, observe, act, and then evaluate the results of the action. Autonomous agents will be covered in Chapter 6 but are still not widely used in production at the

time of writing. This practice of self-reasoning agents is still early and prone to errors, but there are promising signs this approach can be useful in achieving complex tasks, and is likely to be part of the next stage in evolution for AI systems.

There is an AI battle occurring between large tech firms like Microsoft and Google, as well as a wide array of open source projects on Hugging Face, and venture-funded start-ups like OpenAI and Anthropic. As new models continue to proliferate, they're diversifying in order to compete for different segments of the growing market. For example, Anthropic's Claude 2 had an 100,000-token context window, compared to GPT-4's standard 8,192 tokens. OpenAI soon responded with a 128,000-token window version of GPT-4, and Google touts a 1 million token context length with Gemini 1.5. For comparison, one of the Harry Potter books would be around 185,000 tokens, so it may become common for an entire book to fit inside a single prompt, though processing millions of tokens with each API call may be cost prohibitive for most use cases.

This book focuses on GPT-4 for text generation techniques, as well as Midjourney v6 and Stable Diffusion XL for image generation techniques, but within months these models may no longer be state of the art. This means it will become increasingly important to be able to select the right model for the job and chain multiple AI systems together. Prompt templates are rarely comparable when transferring to a new model, but the effect of the Five Prompting Principles will consistently improve any prompt you use, for any model, getting you more reliable results.

Summary

In this chapter, you learned about the importance of prompt engineering in the context of generative AI. We defined prompt engineering as the process of developing effective prompts that yield desired results when interacting with AI models. You discovered that providing clear direction, formatting the output, incorporating examples, establishing an evaluation system, and dividing complex tasks into smaller prompts are key principles of prompt engineering. By applying these principles and using common prompting techniques, you can improve the quality and reliability of AI-generated outputs.

You also explored the role of prompt engineering in generating product names and images. You saw how specifying the desired format and providing instructive examples can greatly influence the AI's output. Additionally, you learned about the concept of role-playing, where you can ask the AI to generate outputs as if it were a famous person like Steve Jobs. The chapter emphasized the need for clear direction and context to achieve desired outcomes when using generative AI models. Furthermore, you discovered the importance of evaluating the performance of AI models and the various methods used for measuring results, as well as the trade-offs between quality and token usage, cost, and latency.

In the next chapter, you will be introduced to text generation models. You will learn about the different types of foundation models and their capabilities, as well as their limitations. The chapter will also review the standard OpenAI offerings, as well as competitors and open source alternatives. By the end of the chapter, you will have a solid understanding of the history of text generation models and their relative strengths and weaknesses. This book will return to image generation prompting in Chapters 7, 8, and 9, so you should feel free to skip ahead if that is your immediate need. Get ready to dive deeper into the discipline of prompt engineering and expand your comfort working with AI.

About the Authors

Mike Taylor cofounded a 50-person growth marketing agency called Ladder with offices in the USA, UK, and EU. More than 400,000 people have taken his marketing and AI courses on LinkedIn Learning, Udemy, and Vexpower.

James Phoenix builds reliable data pipelines for marketing teams, automating thousands of recurring tasks. He has taught 60+ Data Science bootcamps for General Assembly and partnered with Mike on the Udemy course, and Vexpower.

Both authors been experimenting with prompt engineering since the GPT-3 beta in 2020. They slowly automated every part of their jobs with AI and now work as prompt engineers on various projects.

Colophon

The animal on the cover of *Prompt Engineering for Generative AI* is a screaming hairy armadillo (*Chaetophractus vellerosus*). This species of armadillo gets it name due to its habit of squealing, or screaming, when it is handled or threatened.

The screaming hairy armadillo resides in arid areas, specifically in regions in Argentina, Bolivia, and Paraguay. This animal prefers subtropical or tropical regions such as dry forests, scrubland, grassland, and deserts. White and light brown hair cover the animal's limbs and belly. A caparace, a thick armor made of keratin, covers the animal's body, a shield covers its head, and a small band exists between its ears. The animal typically reaches 12 to 22 inches in length, including its tail, and weighs less than 2 pounds, with male armadillos generally being larger than females.

The screaming hairy armadillo is an omnivore, eating small vertebrates such as frogs, toads, lizards, birds, and rodents, as well as fruits and vegetation. It can go long periods of time without drinking water.

Although the IUCN Red List designates the screaming hairy armadillo as Least Concern, it is heavily hunted in parts of Bolivia for its meat and carapace. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on an antique line engraving from *Beeton's Dictionary*. The series design is by Edie Freedman, Ellie Volckhausen, and Karen Montgomery. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.