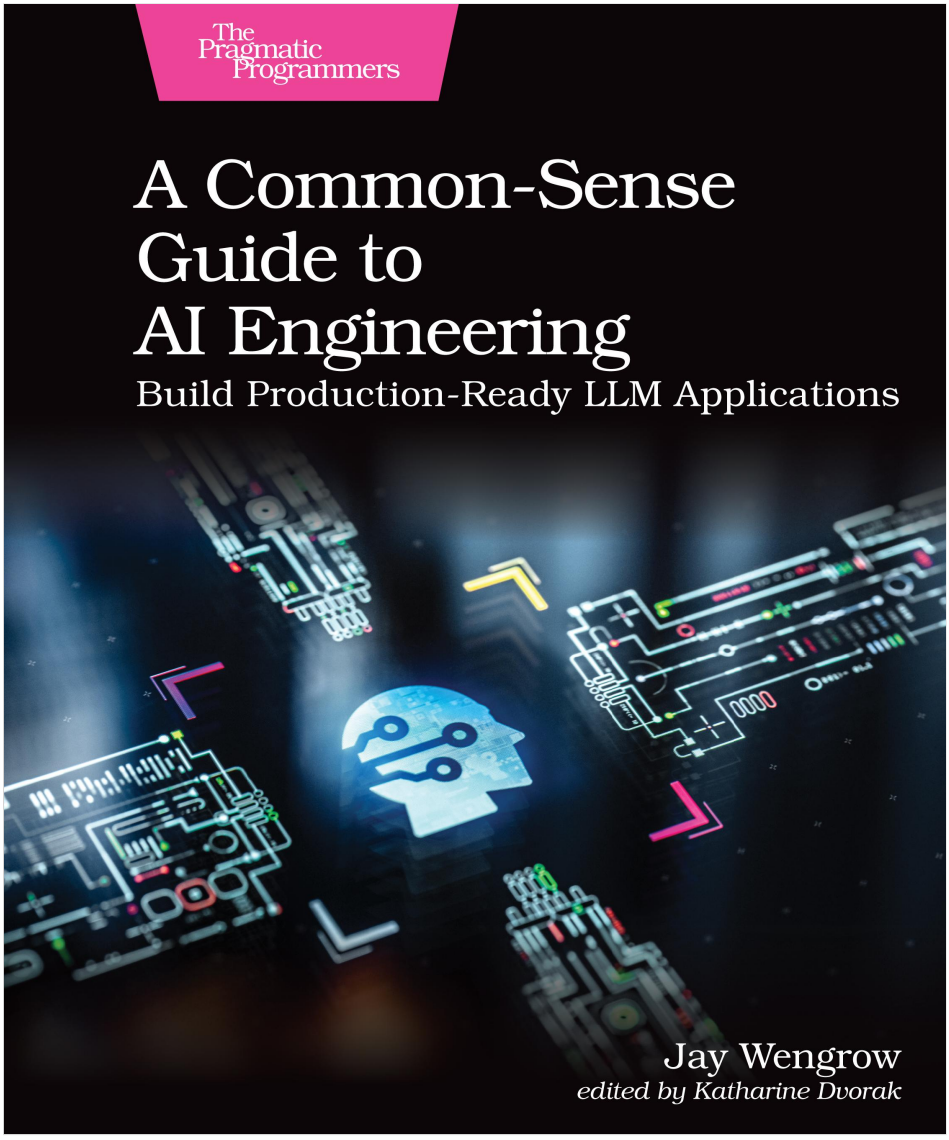


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

CHAPTER 1

HeLLMo, World!

Large language models (LLMs), with their ability to mimic human conversation, are powerful beasts—and the software we build on top of them harnesses all that power. That's what makes AI engineering so exciting. In this chapter, we jump right in and build our very first LLM-powered app. In Chapter 2, Understanding How LLMs Work, on page ?, you'll learn how LLMs work under the hood. But for now, let's get a taste for what working with an LLM is like.

The app we'll create is simple, yet even simple LLM apps are quite capable. We'll start by signing up for an LLM-as-a-service and obtaining an API key. After that, we use the OpenAI Python SDK to write a basic AI application. By the end of the chapter, you'll have a better understanding of why LLMs have taken off like a rocket in the past few years and why they often serve as the engine for software that seems to think on its own. Let's get started!

Signing Up for an LLM-as-a-Service

Throughout this book, we'll use LLMs from OpenAI. There is no single "best" LLM—today's leader may be surpassed tomorrow—but we need a consistent provider. OpenAI, widely credited with bringing AI into the mainstream with ChatGPT, offers a well-documented and reliable API, making it a solid choice for learning the essentials of AI engineering.

You can search the web for "OpenAI API" or head straight to the OpenAI API platform website[1] to sign up for an account.

Next, you'll need to purchase API credits. Choose the minimum option, as the few dollars you'll spend should be more than enough to run all the code in this book and then some.

1. https://platform.openai.com/signup

After going through all the setup steps, which include obtaining an API key, you'll find yourself at the OpenAI platform dashboard. There are a number of useful features here, including links to the API documentation, something you should refer to a lot. (Take a minute to click through it.) As of this writing, OpenAI has a set of more general docs,[2] and then a more specific API reference.[3] Additionally, you can navigate to the "Billing" tab within your personal settings to check how many dollars of credit you still have remaining.

Note that the OpenAI API can be accessed through raw HTTP web requests, but is also supported by SDKs in various programming languages. We're going to use the Python SDK throughout this book.

With your API key in hand, you can now access OpenAI's models. So, let's start writing some code!

Creating Our First App

I'll use the uv Python library to manage dependencies for the projects in this book, but feel free to use another approach if you prefer. You can learn more about and install uv by visiting the official project website.[4]

Let's initialize our new project with:

```
uv init hello_world
```

This creates a hello_world directory with some initial project files. After navigating to that directory, add the dependencies we'll need for this project:

```
uv add python-dotenv
uv add openai
```

In this book, I'll use the python-dotenv library to help store and load the API key effectively. To follow along, create a new file called .env in your project folder and add the following line:

```
OPENAI_API_KEY=your_api_key_goes_here
```

When we initiate our new project with uv, it creates a folder. Within this folder, create a file chatbot.py, which is where we're going to place our code for this chapter. Although we won't build an actual chatbot just now, our code will evolve in future chapters so that our app will indeed become a full-fledged chatbot.

2. https://platform.openai.com/docs/overview
3. https://platform.openai.com/docs/api-reference/introduction
4. https://docs.astral.sh/uv

Fill in chatbot.py with the following code:

hello_world/chatbot.py

```
from dotenv import load_dotenv
from openai import OpenAI

load_dotenv()

llm = OpenAI()

response = llm.responses.create(
    model="gpt-4.1",
    temperature=0,
    input="Who was the first person to land on the Moon?"
)

print(response.output_text)
```

Let's walk through this step by step.

After importing our dependencies, we use the load_dotenv command from the python-dotenv library to load the environment variables into our program. The OpenAI SDK automatically detects whether we have an environment variable named OPENAI_API_KEY, and if we do, the SDK uses that key. This offers the convenience of us not having to write any code explicitly referring to that environment variable.

Next, we create an llm object using the OpenAI SDK. For now, you can think of the llm as representing the actual LLM our code will interact with.

After that, we send text over to the LLM using the responses.create command, specifically placing the text inside the input parameter. The text we send to an LLM is called a *prompt*. In this case, the prompt is the question: Who was the first person to land on the Moon?

In addition to the input, you'll see that we specify a few other parameters of the responses.create function.

Because OpenAI offers several different LLMs, we must specify which LLM we want to use. By setting the model parameter to "gpt-4.1", we ensure that our prompt gets sent to OpenAI's GPT-4.1 model. Soon, you'll see what happens when we swap this for another model.

Another important parameter here is the *temperature*, a concept that will be explained at length in Gauging the Temperature, on page ?. Many LLMs allow you to set the temperature within a range of 0 to 2. The temperature could be set to 0.4, or 1.3, for example.

What you should know for now is that the greater the temperature, the more unpredictable the LLM's output. Now, unpredictable output can sometimes

be helpful for creative work or brainstorming. However, given that our app's goal is to give factual information (about the first Moon landing), we don't want our AI to get *too* creative and possibly start making up false information. The `temperature` in our code is set to `0`, which is common among apps that dispense facts.

Finally, we print out the `response.output_text`, which is the text generated by the LLM. The actual `response` object contains lots of other useful info as well, but the `output_text` is the actual text you'd display to the user in your app.

When I run this program using the command `uv run chatbot.py`, I get the result:

```
The first person to land on the Moon was **Neil Armstrong**. He set foot
on the lunar surface on **July 20, 1969**, during the **Apollo 11** mission.
His famous words as he stepped  onto the Moon were:  *"That's one small
step for [a] man, one giant leap for mankind."*
```

Awesome! We've successfully used code to send a prompt to an LLM and print out a response. Let's run the code a second time, and see what we get. Here's what I got:

```
The first person to land on the Moon was **Neil Armstrong**. He set foot
on the lunar surface on **July 20, 1969**,during the **Apollo 11** mission,
and famously said, "That's one small step for [a] man,
one giant leap for mankind."
```

Interesting! This is pretty much the same result we got before, but it's not *exactly* identical. This foreshadows an important idea we'll learn about in the next chapter, which is that LLMs—even with a temperature of `0`—don't have predictable outputs. You can send an LLM the same request repeatedly, and get a different result each time.

Accordingly, I'd expect that when *you* run the code you may also get slightly different responses than those above. Try it out for yourself!

Tweaking the Model and Temperature

Let's have a little fun and see what happens when we change some of the parameters of the `responses.create` function.

Trying a Different Model

The earlier example uses the `"gpt-4.1"` model, but let's try out some others. The OpenAI documentation[5] lists the various models available. The documentation doesn't always do a great job in making the pros and cons of each model

5. https://platform.openai.com/docs/models

super clear, but if you click around and read the docs enough, you'll get an idea of each LLM's specs. I recommend you do this to get familiar with reading LLM specs in general.

In any case, let's try on a few for size.

As of this writing, there are "cost-optimized" models available, including one called GPT-4.1 mini. When I click into that model's spec page and scroll down a bit, I'll see that the corresponding `model` string for this model is `"gpt-4.1-mini"`. Let's go ahead and replace the `model` parameter accordingly:

```
response = llm.responses.create(
  model="gpt-4.1-mini",  # updated the model to gpt-4.1-mini
  temperature=0,
  input="Who was the first person to land on the Moon?"
)
```

When I run this new version of the code, I get a similar answer as before:

```
The first person to land on the Moon was Neil Armstrong. He set foot
on the lunar surface on July 20, 1969, during NASA's Apollo 11 mission.
```

Technically, this is a win, since this LLM is cheaper than the first model we tried, and we still got an answer of similar quality. However, we've only tested out a single, simple prompt. For other prompts, the two models may diverge more in their responses.

Let's now try out one of OpenAI's "reasoning" models, discussed further in Reasoning Models, on page ?. Specifically, we'll use `"gpt-5-mini"`:

```
response = llm.responses.create(
  model="gpt-5-mini",  # updated the model to gpt-5-mini
  temperature=0,
  input="Who was the first person to land on the Moon?"
)
```

When running this code, I receive an error informing me that `temperature` is not a parameter supported by the `gpt-5-mini` model. This is an important lesson, since it goes to show that sometimes swapping out models isn't always as simple as we'd hope.

In any case, let's see what happens when we drop the `temperature` parameter entirely:

```
response = llm.responses.create(
  model="gpt-5-mini",
  input="Who was the first person to land on the Moon?"
)
```

Once again, I get a similar answer:

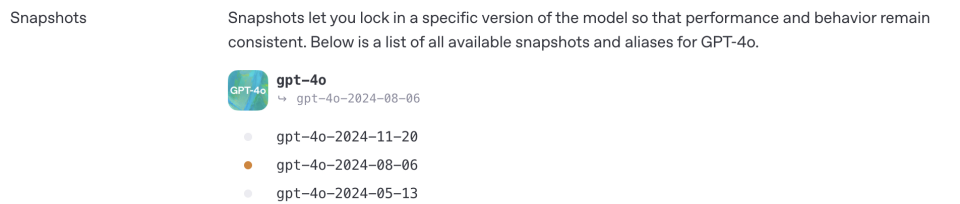
```
Neil Armstrong. He became the first person to walk on the Moon on July 20,
1969, during NASA's Apollo 11 mission.
```

This shows that reasoning models may not provide extra value for some simple prompts.

Now, it *seemed* to me that the response from the gpt-5-mini model took longer to respond than the previous LLMs, but I couldn't be sure. In *the (as yet) unwritten Chapter 17, Observing AI Systems*, I'll demonstrate how to benchmark latency so we can properly compare different LLMs in the realm of speed.

There's a big gotcha to be aware of, which is that a model provider can update an LLM at any time! Sure, when providers update models, they do so for the purpose of making the model better. However, given an LLM's nondeterministic nature, it's possible that a given update can make the model worse for *your* specific application.

To help avoid such a downgrade, some providers like OpenAI provide different versions of each model, and each version is (hopefully) guaranteed to never change. OpenAI in particular calls each version a "snapshot," and you can find all the available snapshots for a given model on that model's spec page. Here are the current snapshots for OpenAI's GPT-4o model:



You can use one of these snapshot strings in your code, like so:

```
response = llm.responses.create(
  model="gpt-4o-2024-11-20",
  temperature=0,
  input="Who was the first person to land on the Moon?"
)
```

For conciseness, we won't generally use a model snapshot in this book's sample code. In real life, though, use a snapshot if you want to avoid heartache. You definitely don't want to find that your app suddenly degraded for no apparent reason.

Changing the Temperature

Okay, now let's add a twist by dialing up the temperature. What response do we get if we set the temperature to 2? I do this here using the GPT-4.1 model:

```
response = llm.responses.create(
  model="gpt-4.1",
  temperature=2,  # updated the temperature to 2
  input="Who was the first person to land on the Moon?"
)
```

In my case, I actually received a very similar response as before, too similar to bother listing here. It seems that when it comes to simple, fact-seeking prompts, even a higher temperature will still yield straightforward answers—at least for GPT-4.1.

To see better how temperature matters, let's change up the prompt ever so slightly to "Who was the first pineapple to land on the Moon?" (Note the use of the word pineapple.)

With a temperature of 0, this was the LLM's response:

```
No pineapple has ever landed on the Moon! Pineapples are tropical fruits
and do not travel to space. The first *person* to land on the Moon was **Neil
Armstrong**, an American astronaut, on July 20, 1969, during the Apollo 11
mission. If you meant something else or are making a joke, let me know!
```

That's true, all right. But here's what happens when I dial the temperature up to 2:

```
It seems there is some playful language—or perhaps an intentional metaphor—in
your question!

There have been **no pineapples** (the fruit (*Ananas comosus*) or
representatives *as pineapples*) to *land on the Moon*.

If you're referencing humans or vehicles that have "weird nicknames," you
might be joking about the call signs or code-names the Apollo astronauts used
(they called their lunar landers by names creation-el513));wego xc
command-forward offline---  yor901),( like *Eagle But—no scientist, astronaut,
embarrassing, nor anything bearing the _spines of psyched tourist stacked
friction sophistication fueled sugary compartersized coffins teammates
delight'l**oda beach), prav bombogenicfølgeligλιαfeeding woolflow robo boost ID
fruitsillat belg pratoleenspielecraft floatingronics snapF behaviouralsk884
friends hasidlefficient businessleave corposa predatorlunCursorBa15leg
puамызmodern gigantic shades:strC cherry routines datasetipad момента
dessa84][$ animais jerseys stages кат постро  байceptors)>suite drifting
cowsbecue anot successorSuper lips...
```

Well, *that* was weird. Talk about an unpredictable answer! I thought it might have come up with a creative answer, but it actually started with the facts,

and then lapsed into some complete gobbledegook. I'm only showing a partial response, since the actual response was much longer.

To that end, there's another really useful parameter called `max_output_tokens` we can add to our `responses.create` call. For example:

```
response = llm.responses.create(
  model="gpt-4.1",
  temperature=2,
  max_output_tokens=500,
  input="Who was the first person to land on the Moon?"
)
```

You'll have to experiment to find the right value for you, but something like this will cut the LLM short in case it goes off the rails and doesn't stop jabbering. That is, it'll just stop generating text after it already produced 500 *tokens*. I'll explain exactly what tokens are in Diving into Tokens, on page ?, but for now think of them as being roughly equivalent to words.

Temperature will certainly matter for "creative" prompts like story generation, such as `"Tell me a story about the first pineapple to land on the Moon."` When I tried this out, the stories generated at a temperature of `1.6` definitely seem more interesting than the ones generated at `0` temperature. Try it out for yourself with your own creative prompts and see what you think.

Checking API Usage

Yikes! In all the fun I've been having in testing out LLMs for this chapter, I've forgotten that each API call I make costs money. Let me run over to the Usage section[6] of the OpenAI Platform dashboard:

Oh, phew. I've made 34 requests using various models (beyond the requests shown in this chapter), and yet my bill so far has added up to all of 9 cents.

6. https://platform.openai.com/usage

Note that this dashboard shows that my "total tokens" is 618, but this refers only to the *input* tokens. *Input tokens* are the text I send to the LLM via prompts, while *output tokens* are the text that the LLM generates in response. LLM providers charge for *both*; they charge for the text the LLM reads, and for the text that the LLM generates.

When I hover over the "total tokens" section, it reveals that the LLM has generated close to 14,000 output tokens, which is certainly the lion's share of my 9-cent bill. Output tokens are usually more expensive than input tokens; check out each model's pricing to see the details.

An important takeaway here is that input tokens cost money as well. Therefore, assuming all else is equal, you'll want to keep your prompts shorter rather than longer.

Wrapping Up

Well, we have our first AI-powered app up and running. The app sends a prompt to an LLM, prints the model's response, and then terminates. The app is as capable as the LLM powering it, which is pretty capable! The LLM knows about the first moon landing, plus a boatload more.

Now that we've gotten our feet wet building an AI-powered app, let's dive deeper into how LLM's actually work under the hood. In order to build *robust* and truly *effective* LLM-powered apps, you have to understand what makes these models tick.