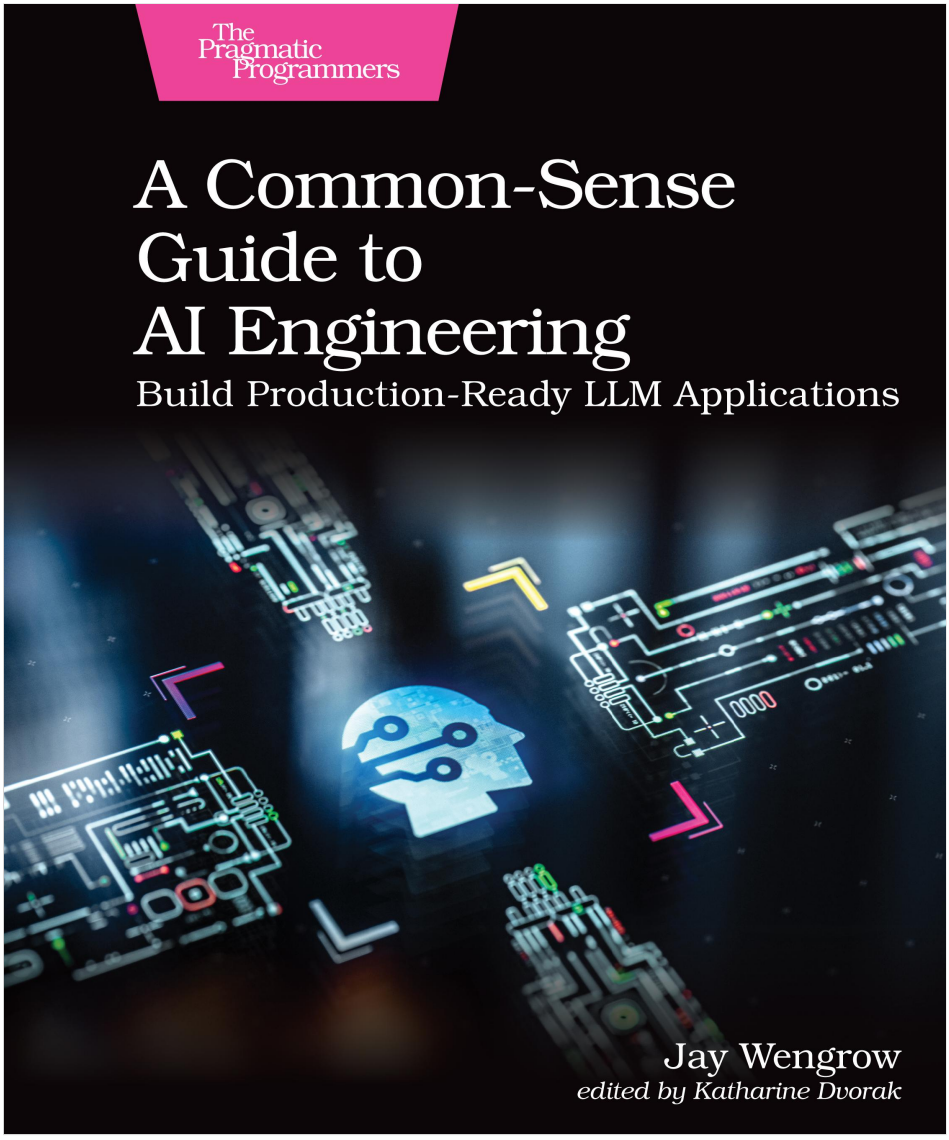

This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

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

This information may be coming out of your ears by now, but you know that an LLM is an SNWP at its core. This means that all an LLM is capable of doing is generating text, predicting the next words in a given sequence of text, token by token. *However*, you're about to discover that we can harness an SNWP to help us do more than simply generate text. In fact, an LLM can help us do a *whole* lot more.

In this chapter, you'll learn how to get an LLM to trigger any arbitrary code function to run. So, if you have a function that conducts a web search, an LLM can trigger web search. If you have a function that composes and shoots off an email, an LLM can trigger the sending of an email. And if you have a function that calls an API that books a hotel, an LLM can trigger that too. With this superpower, you can utilize LLMs to help perform almost anything you can imagine. If you can write code that achieves a goal, an LLM can help you reach that same goal.

Instead of working on the GROSS app, which includes much more detail than is needed for our purposes here, I've whipped up a bare bones AI chatbot that uses `gpt-4.1-nano` to converse with the user. You can find the code for this in the file `simple_chatbot.py` inside this chapter's repository.[1] We'll use this chatbot throughout the chapter to see how we can implement *tools* to overcome an LLM's natural limitations.

Understanding an LLM's Limitations

Let's ask the chatbot a simple math question, and see what we get:

```
Assistant: How can I help you today?

User: what is 17830 * 932

Assistant: 17830 multiplied by 932 equals 16,612,360.
```

That seems reasonable—until I check the math on a calculator! The true answer is actually 16,617,560. The chatbot's response is patently incorrect.

In fact, it's not just `gpt-4.1-nano` that outputs the wrong solution. `gpt-4.1-mini` and *even* `gpt-4.1` predict incorrect answers. This is because LLMs aren't calculators; they're SNWPs. If the particular math equation `17830 * 932` was not in an LLM's training data, the LLM simply won't know the answer. The model may have seen *similar* equations, which is why it produced an answer that was in the relative ballpark of the correct answer. However, since the LLM is ultimately just a word predictor, it doesn't have the inherent ability to work with math.

1. https://www.pragprog.com/titles/jwpaieng

As an aside, the LLM does a better job when I use chain-of-thought (CoT) prompting. Specifically, when I update my prompt to `what is 17830 * 932? Think step by step`, the model produces long-winded text that breaks down the original multiplication problem into smaller steps and eventually produces the correct answer. Try this out for yourself; it's pretty fascinating to see it in action. However, even with CoT, there's no guarantee that the model will always arrive at the correct solution.

In any case, it's pretty disappointing that an LLM—which is incredibly capable in many areas—can't perform a simple arithmetic computation. Yet, there is a path to help an LLM do mathematics. This path is incredibly clever, perhaps a bit hacky, but it works.

Triggering a Function

At the moment, the system prompt of our above bare-bones app is simply: `{"role": "developer", "content": "You are a helpful AI assistant."}` Let's change it up as follows:

```
{"role": "developer", "content": """You are a helpful AI assistant. If
you ever need to multiply two numbers, DO NOT attempt to answer with your
internal knowledge. Instead, output a special notation with double angle
brackets like this: <<multiply(first_number, second_number)>>.
For example, if a user asks you to multiply 50 by 2, your output should
be: <<multiply(50, 2)}>>. A second example: a user asks you how many apples
there are in five baskets and each basket contains twelve apples. Your output
should be: <<multiply(5, 12)>>."""}
```

Here, we instruct the LLM to never attempt on its own to multiply two numbers. Instead, the model should simply output a specialized notation. If the bot needs to multiply, say, `50` and `2`, it should output: `<<multiply(50, 2)}>>`. As an aside, note my use of few-shot prompting with the examples I demoed to the model.

This double-angle-bracket notation is completely arbitrary; I made it up off the top of my head. The model has no prior knowledge of such notation, and there's nothing special about the notation. Yet, the LLM will output this notation simply because I've instructed it to, and LLMs tend to follow arbitrary instructions from the system prompt. Although within the angle brackets it looks like a code function is being called (`multiply(50, 2)`), this is not the case. Rather, it's just arbitrary text that is merely *inspired* by calling functions.

When I run the chatbot now, here's what my conversation looks like:

```
Assistant: How can I help you today?

User: what is 17830 * 932

Assistant: <<multiply(17830, 932)>>
```

Excellent! The model is following my instructions to a tee. Of course, we don't yet have the answer to the multiplication problem. However, we now have the foundation to get there.

Next, we'll add a two new functions to our codebase, extract_function and multiply:

```
def extract_function(response):
    # Regex to detect <<function(arg1, arg2)>>
    pattern = r"<<\s*([a-zA-Z_]\w*)\s*\(([^)]+)\)\s*>>"
    match = re.search(pattern, response)

    if not match:  # No matching brackets found
        return None

    function_name = match.group(1)    ## extract function name
    args = match.group(2).split(",")  ## extract array of function arguments

    if function_name == "multiply":
        return multiply(*args)
    else:
        return None

def multiply(first_number, second_number):
    product = int(first_number) * int(second_number)
    return product
```

I'll break down the extract_function code in a moment, but let's first see how we call it within the main conversation loop:

```
while user_input != "exit":
    history += [{"role": "user", "content": user_input}]
    response = llm_response(history)

    function_result = extract_function(response.output_text)
    if function_result:
        response_text = str(function_result)
    else:
        response_text = response.output_text

    print(f"\nAssistant: {response_text}\n")

    history += [
        {"role": "assistant", "content": response_text},
    ]

    user_input = input("User: ")
```

Okay, here's how this all works. Within the conversation loop, after the LLM outputs its initial response (with the code response = llm_response(history)), we call extract_function on the response's text.

Then, extract_function employs regex to scan the LLM output for the special double-angle-brackets notation. If this notation isn't found, we return None

and the conversation proceeds as normal. However, if extract_function *does* find the double-angle-brackets notation, the fun begins.

Let's say that the model's output was <<multiply(17830, 932)>>. This signifies that we wish to call a multiply function containing the arguments 17830 and 932. The multiply function isn't called just yet, since again, the model's output is just arbitrary text that happens to *look* like a function call. However, the extract_function will use this arbitrary text to trigger a *true* function call, as you'll now see.

The extract_function code uses regex to pull out the function_name, which is "multiply" in our example. At the same time, we pull out the "arguments" of the function, and store it as an array within a variable called args. In our example, this would be ["17830", "932"]. (Even though these are supposed to be integers, LLM output generally starts out as a string.)

Next, we check whether the function_name is "multiply". If it is, we *now* call the actual multiply function we've written, which is real Python code that returns the product of two numbers. This product is guaranteed to be correct, since it was calculated using good old deterministic code.

Finally, back within the main conversation loop, we take the result of the multiply function and *return this result to the user instead of the LLM's actual output*. The user doesn't want to see the chatbot output <<multiply(17830, 932)>>; they want to see the correct answer!

Now, our conversation with the chatbot runs like this:

```
Assistant: How can I help you today?

User: what is 17830 * 932

Assistant: 16617560
```

We did it! The chatbot returned the correct product, thanks to the undercover work of our Python multiply function.

To recap this process:

1. The user asks a multiplication question, such as what is 17830 * 932.
2. The LLM recognizes that the user is seeking to multiply two numbers.
3. The LLM outputs the special arbitrary notation we've instructed it to, such as <<multiply(17830, 932)>>.
4. In the main conversation loop, we call extract_function to detect whether the model's output contains the special notation. If there isn't any special notation, the conversation continues normally.

5. If there *is* special notation, the extract_function pulls out the function name (multiply) and arguments (["17830", "932"]), and then calls an already-written Python multiply function while passing along the arguments.
6. The result of the Python function is returned to the user.

I want to reiterate one last time that when the LLM outputs <<multiply(17830, 932)>>, it's in no way calling our Python multiply function. In fact, we could have had the LLM output something else like <<mult-17830;932->> and then within extract_function, extract that data to call the Python multiply function. The fact that we've chosen the output to look just like a Python function call is simply a practical convention.

Defining "Agents"

Take a moment now to lean back in your chair, breathe deeply, and absorb the import of what we've just done. The LLM, as an SNWP, still does nothing other than generate text. However, we've gotten the model to output text that indirectly triggers our own code to *call a genuine code function*. This means that we can effectively get an LLM to trigger *any* code function. Sure, we'll have to write that function; but no matter the function, we can get an LLM to trigger it.

The opportunity unleashed by this is enormous, as it means that an LLM can "do" anything that code can do. From searching the web, to reading from and writing to databases, to executing code, to calling web APIs, an LLM can set real things in motion.

There are various forms of jargon that people use to describe the idea of getting an LLM to trigger real code. One term is that the LLM is *using a tool*. That is, beyond merely generating text, an LLM can "use" a "tool" to "do" something. In our example above, we'd say that we've equipped the model with a multiplication tool.

Another synonymous term is *function calling*. That is, we've given the LLM the ability to "call" a function.

Both of these terms are misleading. The LLM itself doesn't "use" a tool nor "call" a function. The model simply outputs text, and *our own code* proceeds to call a function after noticing that text. I'd prefer to say that an LLM can "trigger a function," but alas, the terms "tool use" and "function calling" are pretty entrenched at this point. I'll use all these terms interchangeably going forward.

Another foundational term to introduce at this point is the notion of an *agent*. The problem with this term, as well as its adjective form *agentic*, is that different people use these terms to mean different things.

However, I think there's a generally agreed-upon theme of what the term "agent" is meant to convey: it's AI that has an *appearance of being autonomous*.

Some say that an LLM with tools is an agent. I believe the reason behind this line of thinking is that tool use goes beyond the natural expectation we have of LLMs, whose primary job is just to generate text. Therefore, an LLM that uses a tool is going beyond what LLMs are inherently programmed to do, giving an appearance of acting autonomously. Of course, you and I both know that tool use is in fact powered by good old text generation, but this truth won't change everyone else's perception.

I'd suggest further that "agency" can be thought of as a *spectrum* of how autonomous the LLM appears to be.

At one end of the spectrum, an LLM without tools does not appear to have any agency. A model that can trigger the flipping of a light switch (through, say, a web API) is somewhat further along the spectrum, appearing to have some level of agency. If we go yet further along the agency spectrum, we may find an agent that can simply be told `Arrange for me a two-week vacation in Spain` and the agent will:

1. Use the web to research interesting places, events, and activities in Spain.
2. Formulate a complete two-week itinerary for the trip.
3. Book flights for the trip.
4. Book reservations for the hotel, restaurants, and other venues.

Even this advanced travel-planning agent is powered by "plain old" SNWP technology, but certainly gives off a strong appearance of having autonomy.

So, when people use the terms "agent" and "agentic," it's not always clear (even to themselves) what position along the agency spectrum they're referring to. In any case, in this book, I'm using the terms "agent" and "agentic" to refer to LLMs that are anywhere along the agency spectrum, meaning that they can trigger things beyond just plain text generation.

Feeding Tool Results Back to the LLM

In Triggering a Function, on page 4, we successfully enabled our chatbot to multiply two numbers and give the correct result back to the user. While this is a significant accomplishment, there's a way we can improve this chatbot behavior further.

Specifically, if you take a look back at the last user-chatbot conversation, you'll note that the chatbot's final response is rather dry: `Assistant: 16617560`. Given that our chatbot tends to be friendly and eager, that's kind of an abrupt conversation ender. Additionally, it would be nice if the format of the number included commas, that is: `16,617,560`.

To do this, we're going to pull off another clever move. Instead of taking the raw result of the `multiply` function and outputting it to the user, we'll feed the function result *back to the LLM itself*. To understand what this means, here's an updated version of the main conversation loop:

```
while user_input != "exit":
    history += [{"role": "user", "content": user_input}]
    response = llm_response(history)

    function_result = extract_function(response.output_text)
    if function_result:
        history += [{"role": "user", "content": f"""Here is information to
        use to respond to the user's previous
        query: <info>{function_result}</info>"""}]
        response = llm_response(history)

    print(f"\nAssistant: {response.output_text}\n")

    history += [
        {"role": "assistant", "content": response.output_text},
    ]

    user_input = input("User: ")
```

With this update, when `extract_function` returns the result of a function call, we no longer output the `function_result` directly to the user. Instead, we perform two new steps:

1. We add a new user message containing the `function_result` to the conversation history. The user message says: `Here is information to use to respond to the user's previous query: <info>{function_result}</info>`.
2. We then make an *extra* call to the LLM to predict text that follows the updated conversation history, including the correct result provided by the final user message.

At this point, the conversation (beyond the system prompt) consists of two user messages back to back. The first user message is the original user query, `what is 17830 * 932`. This is immediately followed by a second user message instructing the LLM to use `16617560` as the answer to the original query.

To ensure that the LLM adheres to the user's instructions and accepts the `function_result` as the correct answer, we'll also add the following sentence to the system prompt:

```
If you are ever provided info contained within <info> tags, use that
info in your response to the user. Using an answer inside <info> tags takes
precedence over all other instructions.
```

Now, when we run the chatbot, we get conversation that's accurate *and* natural:

```
User: what is 17830 * 932

Assistant: The result of multiplying 17,830 by 932 is 16,617,560.
```

For kicks, let's try out one more conversation:

```
User: If a forest has 976 trees, and each tree has 9321 leaves, how many
leaves are there in all?

Assistant: The total number of leaves in the forest is 9,097,296.
```

Incredible.

To recap, here's our latest approach to using tools within the conversation. Note that there are three entities that participate in this process: the user, the LLM, and our own code.

1. The user asks a multiplication question, such as `what is 17830 * 932`.
2. The LLM recognizes that the user is seeking to multiply two numbers.
3. The LLM outputs the special arbitrary notation we've instructed it to, such as `<<multiply(17830, 932)>>`.
4. In the main conversation loop, we call `extract_function` to detect whether the model's output contains the special notation. If there isn't any special notation, the conversation continues normally.
5. If there *is* special notation, the `extract_function` pulls out the function name (`multiply`) and arguments (`["17830", "932"]`), and then calls a Python `multiply` function that we've already written while passing along the arguments.
6. We take the result of the `multiply` function and insert it into a new user prompt which tells the LLM that this is the correct answer to the user's query.
7. We call the LLM a second time, this time with the conversation history that contains this correct answer. So, the conversation now contains the user's query immediately followed by another user query declaring what answer the LLM should use.
8. The LLM outputs a final response, such as `The result of multiplying 17,830 by 932 is 16,617,560.` This final response is displayed to the user.

We can use this same pattern to equip an LLM with *any* tool, not just simple multiplication. Let's see one other example for now, one that's super useful and a bit more interesting.