CHAPTER 10
Evaluating LLM Applications

GitHub Copilot is arguably the first industrial-scale application using LLMs. The curse of going first is that some of the choices you make will seem silly in hindsight, laughably flying in the face of what (by now) everyone knows.

But one of the things we got absolutely right was how we got started. The oldest part of Copilot's codebase is not the proxy, or the prompts, or the UI, or even the boilerplate setting up the application as an IDE extension. The very first bit of code we wrote was the *evaluation*, and it's only thanks to this that we were able to move so fast and so successfully with the rest. That's because, for every change we made, we could check directly whether it was a step in the right direction, a mistake, or a good attempt that just didn't have much of an impact. And that's the main advantage of an evaluation framework for your LLM application: it will guide all future development.

Depending on your application and your project's position in its lifecycle, different types of evaluation may be available and appropriate. The two big categories here are offline and online evaluation. *Offline evaluation* is evaluation of example cases that are independent of any live runs of your application. Since it doesn't require real users or even, in many cases, an end-to-end working app, it will typically be the evaluation you implement first in your project's lifecycle.

Offline evaluation, however, is somewhat theoretical and possibly a bit disconnected from the real world. But once you deploy your app to the real world, you unlock *online evaluation*, which tests your ideas directly on your users. Being live raises the stakes for online evaluation compared to offline evaluation: you'd better be sure your ideas aren't so terrible as to totally ruin the user experience, and you also need enough users to get sufficiently clear feedback in the first place. But if you overcome these hurdles, then the data you gather will be extremely valid for your use case in a way you can't be sure of with offline evaluation.

Both offline and online evaluations are important, but before we delve into them, let's zoom out for a second and ask ourselves a primary question.

What Are We Even Testing?

Evaluation can assess three things:

- The model you use
- Your individual interactions with the model (i.e., your prompts)
- The way many such interactions fit together in your overall application

Think about the loop that represents one run of your application, which we discussed in Chapter 4. As in traditional software testing, there's a benefit in trying to test both the whole interaction (think regression tests) and the smallest building blocks, which in this case correspond to one pass of the model (think unit tests).

Many application workflows have only a single call to the model, so the distinction isn't very meaningful. But for those applications that have large loops using iterated calls, you'll design a test harness by carving out particular parts of the loop and declaring "This is what I'm testing now!" You're not free in that choice; particular parts will be hard to test, but the ideal would be to have some regression tests that cover as large a part of the feedforward pass of the loop as possible and to also have unit tests for every interaction you deem critical (i.e., interactions that are hard and important).

In all your tests, record total latency and token consumption statistics. While usually not the main focus of evaluation, they are easy to assess, and you'll want to know of any big effects here.

If you have such a suite of tests, you can use them to assess the different components of your app as follows:

- If you're thinking of swapping out the model or upgrading it, you'll probably want to capture as large a part of the app as possible. You can test each unit individually, but going for regression tests that cover a large section of the loop is a bit more natural—unless you're thinking of mixing and matching models (e.g., for cost or latency reasons). In that case, looking at each pass in isolation makes more sense.
- If you want to optimize your prompts or other API parameters like temperature or completion length, your main focus should probably be on the small unit tests that capture a single pass to the model. After all, that's what's affected directly by

a single prompt change. If your regression tests are powerful enough, you can use them as well, but it's easier for statistical noise to drown out the individual effects that would be apparent at the unit level.

- If you're tinkering with the overarching architecture of the whole app (e.g., considering changing the overall shape of the loop), then by definition, regression tests are what you need to compare different approaches.

In sum, all test setups are useful, but if you have to choose one as the most important starting point, it's probably best to have something that tests the whole loop. After all, testing should mirror reality, and in reality, it's the performance of your whole system that you want to optimize for. Once you have a harness that covers (close to) the whole loop, you can still add specific tests for particularly critical parts of the loop.

Offline Evaluation

There's a large range of complexity your offline evaluation suites can have. We found it useful to start out with something simple.

Example Suites

When you write version 0 of your prompts, you'll probably have a window open with an LLM chat, or you may have a completion playground environment, where you try out an example or two. That's not scalable, but there's a scalable version of it that's immensely useful: the example suite. An *example suite* has a simple setup made out of three components:

- A set of 5 to 20 examples of inputs to your application or one of its central steps. If possible, these should span the range of scenarios you'll expect to encounter in reality.
- A script that applies your application's prompt-making to each of the examples and asks the model for the completion, outputting both the assembled prompts and the completion as files.
- A way to eyeball differences among such files, for example, by committing them to your repo and looking at `git diff`s.

An example suite is not like a test suite in the software testing sense (although it could later evolve into one). You'll have no automated way of knowing whether any change is an improvement or a regression. Instead, you'll have to go through the differences yourself and decide whether you consider them improvements or regressions. That's more of an investment than running a test suite and checking the headline result.

But there are two big advantages to this setup. The first is that you can start it the moment you codify your very first prompts, before you have any way of assessing your output at all. The second is that as you become familiar with these examples, you'll not only see whether a new prompting scheme works or doesn't work, but you'll also be able to see typical shortcomings in the completion and decide to adjust your prompts to address them specifically.

For example, we were working on a project at GitHub regarding pull request (PR) summarization. PRs are common elements in software development in which a developer proposes code changes that a reviewer is supposed to check, and we wanted to give them a leg up by summarizing the changes in a small number of bullet points. So we took a set of tens of example PRs (mined from GitHub), and by eyeballing the summaries, we could see the typical problems of our summarizer with different formulations of the prompt. If we thought it was too terse, we could quickly add the word detailed to the prompt and immediately observe the effect. If we thought it was too verbose, we could ask it to limit itself to one or two paragraphs. If it made wild assumptions about the reasons motivating the PR, we could ask it to limit itself to describing the functionality. In fact, we'd ask it for a paragraph about the functionality and a second paragraph about putting the functionality in the context of the project goals, and we'd just not surface that second paragraph, using the trick we discussed when talking about fluff in Chapter 7. By allowing us to easily compare the effects of different prompts, the example suite proved to be an incredibly useful combination: it was systematic enough to alert us to the consequences of changes while being flexible enough to provide value even before we came up with stringent quality criteria.

Example suites are great for directed exploration, but their scale is limited by the number of examples you're willing to eyeball each time you make a change. For subtle effects, however, you'll want many hundreds of examples, maybe thousands. Figure 10-1 illustrates that if you want to unlock the statistical power that comes with such a harness, you need to solve two problems:

1. Where do you get the example problems?
2. How do you assess your app's solutions to those problems?

You can upgrade from playground tinkering to an example suite once you have your first code implementation written out. To graduate to an evaluation harness, you need lots more examples and a way to automatically assess suggestions.

With the word *example*, we're referring to a particular situation in which you might run your app. For simple loops, that's pretty straightforward—if you call the LLM once, then one instance of all the context that could theoretically go into the prompt for that one call is an example problem, and what you could hope to get out of it (after processing) is the example solution.

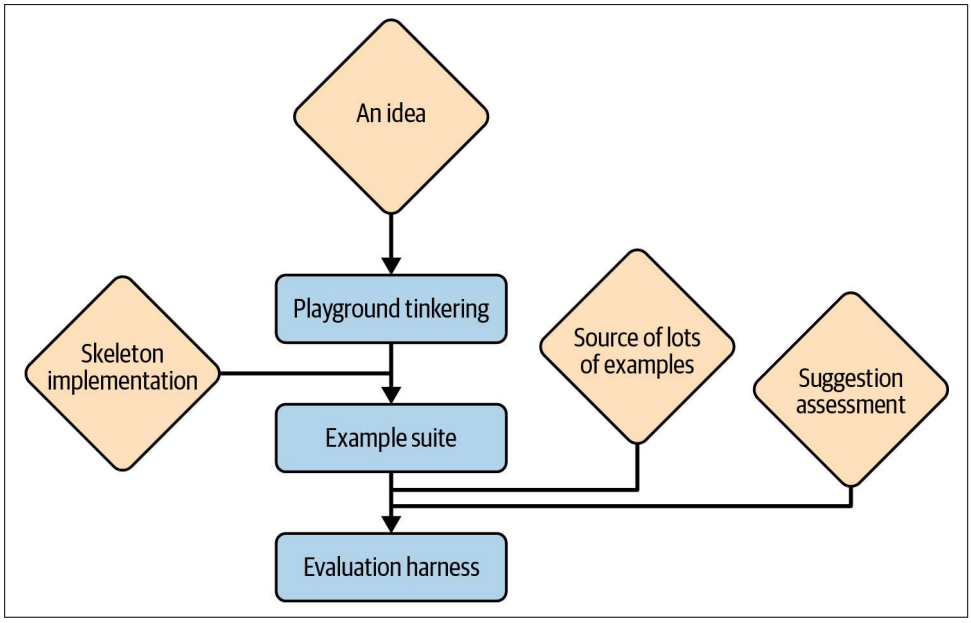

Figure 10-1. The tech tree of offline evaluations

But by now, we know of more complex, interactive architectures where the LLM is called several times, and those calls depend on each other. The most complex case probably occurs when a conversation between user and LLM occurs. There are two options for evaluating such cases:

- You give up on evaluating the whole loop, and instead, you evaluate individual passes of the conversation. For example, you can use so-called *canned conversations*, in which a whole script is written out, and you can evaluate the model on each of its conversation passes on how well it performs at that pass. Then, *regardless of what the model actually answered*, you can move on to test the next step in the conversation by assuming the model had used the answer from the canned conversation instead (see Figure 10-2).
- You can use the model to mock out the user's side of the conversation. In this case, the example consists of a profile of the user, which is a bit like the instructions in improv theater. The model will use that profile to emulate that real user. This allows you to test the whole loop, at the price of possible model shortfalls being baked in—in particular, shortfalls like misunderstandings of the domain or prejudices regarding how users are likely to behave. It's not a perfect method, but often, it's the best you can do.

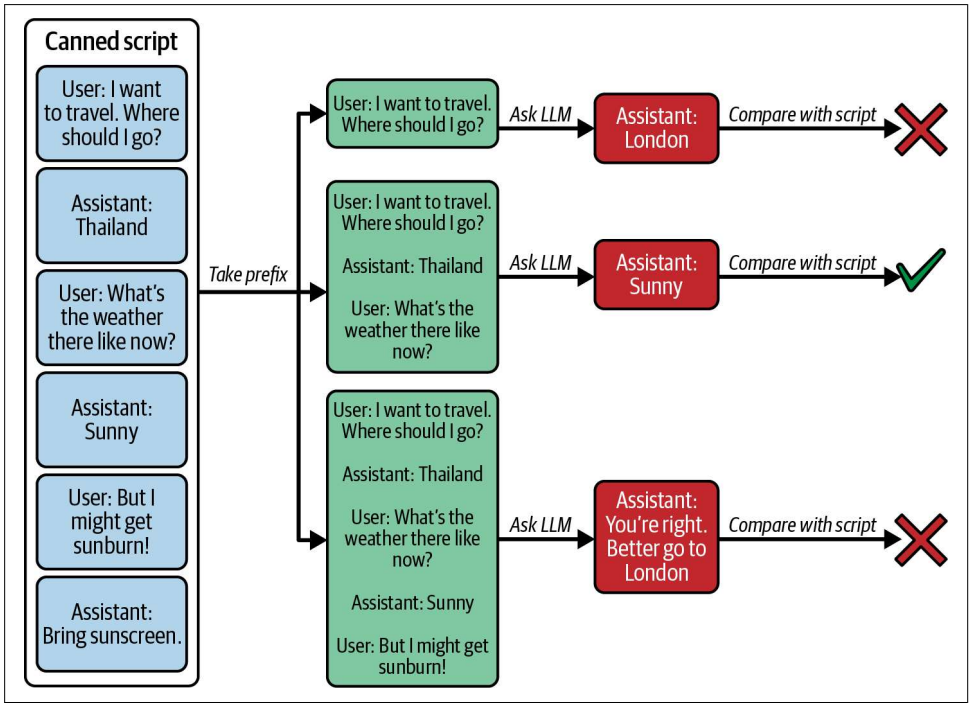

Figure 10-2. Canned conversations

Finding Samples

There are three main sources for the many examples you'll need to find:

- They already exist, and you just have to find them.
- They're being created by your project, and you'll collect them.
- You'll have to make them up entirely.

We'll address each source in turn, starting with what already exists.

Each LLM app solves a specific problem, and that problem (or one of its subproblems) may be similar to one that can be mined, because there are many example/solution pairs out there. If you're lucky, the app you have in mind will solve a problem that users have solved themselves (without AI assistance) thousands of times and that generated records. I recently came across AI assistance in prefilling a summary field for an online form. Whoever programmed that feature most probably had records of tens of thousands of forms where humans had filled out that summary field for themselves. These would have provided a rich source of samples for any evaluation harness.

But very often, what you can mine is only similar, not identical, to the problem your app wants to solve. In that case, your source of samples needs to strike a balance: find a source of samples that's ubiquitous enough in real world corpora to scale but similar enough to your application problem to allow valid conclusions. It's a stepping stone between the lab and reality.

For example, the original GitHub Copilot problem is "What will the user want to type next?" If GitHub Copilot knows the answer to that question, it can then suggest it as grayed-out text while the user is typing. But there are no large-scale open source corpora for exactly this question. What does exist, however, is a large-scale corpus of open source code in the form of all the repositories on GitHub.

So we chose to generate samples by performing these steps:

1. Take an open source repository, and from it, take one code file, and from that file, take one function.
2. Remove the body of that function, imagining that the user was just writing that file and had almost finished writing up everything except the actual implementation for this one function, but their cursor is where the implementation would go.
3. Ask GitHub Copilot what to type next.

This is not identical to the actual, real-world problem, for several reasons. For one, the distribution is skewed: whole function bodies are longer than the typical block suggested by Copilot. For another, any changes to the rest of the file that depend on the body (e.g., imports added to the preamble) have already happened. None of that is ideal, but it balances with the fact that this is a near-infinite well of samples.

But maybe you have thought about it for a long time and have found no existing source of data sufficient for your purpose, neither direct instances of the problem the app addresses nor similar cases. So you need a new data source. Here's the good news: you're currently writing a source of data. The app that you're building is a creator of example cases for its own problem, of course, with new samples accumulating as users use the application. That sort of data is as realistic as possible, of course, but there are also significant drawbacks:

- Data only starts rolling in once your first prototype has rolled out.
- Whenever you make significant updates to your app, there's a good chance your earlier data has become obsolete.
- Recording extensive user telemetry requires very high standards in obtaining consent, handling, and safeguarding of the data.
- Application interaction is a source of great example problems (inputs) but not necessarily of great example solutions (outputs). Even if you can record what

action the user actually ended up taking, that will be very heavily influenced by the action suggested by your app.

We'll see in the following paragraphs that not all evaluations rely on knowing which solution is the single right one (also known as *the gold standard solution*). In this case, gathering data from application interactions can be worthwhile. Otherwise, we'd recommend that you leave telemetry from the app for online evaluation, which avoids a few of the problems surrounding data handling and adds some extra advantages.

So what can you do instead? Well, you can always make stuff up—probably not by hand and probably not all by yourself (this whole section is about scale, after all), but by now, you're an experienced AI developer and can ask the LLM to generate samples. In some cases, this can work amazingly well. That's particularly true in those cases where you can start with the solution and make up the problem from there. Alternatively, if you don't need a gold standard solution at all, just generating situations is something that LLMs excels at. If you go down that route, it's a good idea to go hierarchically, as follows:

- Either ask the LLM to come up with a list of topics or present one yourself. If your problems have several aspects that can be combined, you can make good use of the fact that if you have n options for aspect A, m options for aspect B, l options for aspect C, and k options for aspect D, there are $n \times m \times l \times k$ possible combinations. Exploiting combinatorial explosions like that can easily give you a very large amount of topics that are well distributed over a large space.
- If you want more samples than topics, you can still ask the LLM to come up with several samples per topic. Provided your context window is long enough to output them all, asking for several examples in one go usually leads to a wider variety than just asking the LLM repeatedly with a temperature setting greater than 0 to get several options.

If you're not sure the LLM has complete command over the problem space, the generated examples may well be overly simplistic and exaggerated tropes, may rely on popular misunderstandings, or may simply be incorrect. Even more dangerous is the incestuous relationship between the LLM that tests and the LLM that comes up with the tests—if those two LLMs are one and the same, it biases the outcome. For example, if you're using your test harness to decide whether to switch from model A to model B, if all the samples were made up by model A, then chances are model A will have a leg up on model B.

Each of these approaches to finding samples has advantages and drawbacks, and depending on your particular situation, you may settle on one or more of them. That will give you lots of samples—maybe with gold standard solutions and maybe not. You can run your app on them and get a candidate solution for each of them, but now what?

Evaluating Solutions

If you want to evaluate possible solutions at scale, there are three main approaches. Ordered by difficulty, they are matching the gold standard (whether exact or partial), functional testing, and LLM assessment.

Gold standard

The easiest way, if you can manage it, is matching the gold standard (i.e., an example solution for your example problem that you have some confidence in). For example, if you've mined historical records, it could be what the human without LLM assistance did. Depending on what kind of solution your app offers, this might be all you'll ever need, especially if the solution can be expressed very simply.

In the easiest case, your LLM application is supposed to arrive at one single yes/no answer in the end, and you have some gold standard data of good decisions. Then, all you need to evaluate is to check how often your app's decision matches the gold standard. For example, Albert once worked on an application for unit test generation, and the first step in that app's loop was to ask itself, "Do I even need unit tests for this piece of code?" That's a question with a yes/no answer, and it was easy to validate the app's performance on this step by checking how well it matched gold standard solutions.

Evaluation of binary decisions or multilabel classification using gold standards can just be counting how often the model gets it right. But if you crave more statistical power, you can use logprobs as discussed in Chapter 7.

However, very often, the output of an LLM is more or less free-form text. You can use exact match counts here, tallying how often your app produces a candidate solution that is verbatim the same as the gold standard. But the greater the degree of freedom you have, and in particular, the longer the model's answer is, the rarer exact matches will be, even for great models. At some point, the chance to get an exact match is so low that the metric becomes more or less meaningless. Even before that point, it raises a question: what are you optimizing for, correct solutions or solutions that are formulated in a particular style?

That's where partial match metrics can be useful. They work by picking out one particularly important aspect of the solution and matching only on that. For example, if the LLM is supposed to write source code for you, you may want to ignore comments, blank lines, or (depending on the language) even all whitespace. Therefore, you choose the partial match metric "exact match after deleting all comment lines and removing all whitespace." If the LLM is supposed to suggest travel destinations, you

might want to match on the country of destination but ignore all other details the model gives you: that's another partial match metric.

All partial match metrics come with a hard choice: you need to figure out which aspect of the solution you really care about. That's easier said than done, because in most applications, a catastrophic failure in any aspect of the solution can theoretically invalidate the whole thing. But some modes of failure are more likely, so you can guard against those.

Let's work through an example. Imagine you're writing a smart home manager. You're looking at a situation where the user says, "I'm chilly" (see Figure 10-3). You've already determined a gold standard solution for this: the system could set the temperature to 77°F, and that would be perfect. Checking for a partial match might consist of checking for only whether the manager regulates the right system (in this case, heating). It's sensible to check for that because there's (probably) a real chance the manager doesn't react by adjusting the heating system, and that is likely to be a real failure on its part. On the other hand, if the manager does adjust the heating system, it's likely that it will adjust it to something sensible, whether it's exactly 77°F or not. The manager could set the temperature to 0°F, of course, but that's a less likely failure case when compared with the possibility of the system not understanding at all that it's supposed to regulate heating or not knowing how exactly to regulate heating. So, it makes sense to test for setting *any* temperature, rather than testing for the exact temperature the model should set.

Generally, it's best to evaluate on an aspect that meets these two criteria:

- The aspect is good at distinguishing between breaking and benign divergence from the gold standard solution. This makes the evaluation meaningful or valid.
- The aspect isn't too specific; if it were, the LLM would have little chance of getting it right. The aspect also isn't too general; if it were, the evaluation would be meaningless.

Both criteria require you to play around with the model a bit to see where some typical mistake patterns lie and just how bad those mistakes are. Unfortunately, there's some circularity being introduced here because you choose the test based on what your LLM or setup is *currently* good at, and you'll use that evaluation to guide its *future* development. But that's still much better than choosing a weak or misleading aspect for your evaluation framework.

If the LLM doesn't return purely free-form completions, testing an especially critical one out of the several fields it contains is often a good aspect to focus on for partial match metrics. In particular, that holds for applications that are heavy on tool use: you can check on whether the right tool is used and maybe whether it's called with the right syntax (see Figure 10-3).

Checking on whether the model uses the right tool is also an instance of following another bit of general advice: when the model makes several decisions in a row while outputting its tokens one by one, it makes sense to evaluate the first decision that's got a real chance of going wrong (and to invalidate later decision points). In Figure 10-3, the model first decides to use any tool at all by beginning with `to=functions.`, then commits to the specific tool `set_room_temp`, and then settles on the particular values `{"temp": 77}`.

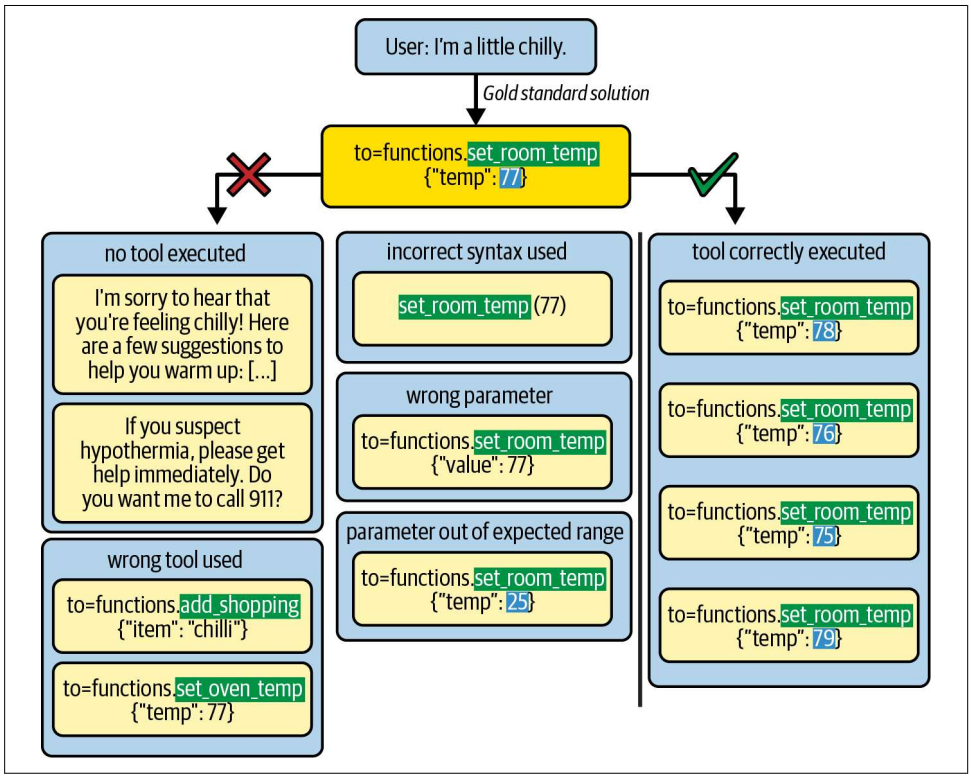

Figure 10-3. Testing that the right tool is called with the right syntax

In the figure, note that if the `set_room_temp` tool isn't called correctly (as on the left), the suggestion is most likely useless. If the tool is called correctly but in a different way than in the gold standard solution (as on the right), the chance that the suggestion is reasonable (as on the upper right) is still substantial.

Functional testing

What if you don't have a gold standard solution or can't easily compare it to your app's solution? One option is *functional testing*: taking the completion and confirming that certain things "work" with it. For example, you can count how often the LLM

gives you a completion that you can parse, calls only the functions and tools you have available (and with arguments of the right types), etc. In most applications, that is too weak, but occasionally, you can go pretty far with functional testing.

Let's look again at the Copilot evaluation framework as an example of such functional testing. The evaluation framework would simulate cases where Copilot was used to reimplement a function from an open source repository, and then it would check whether the unit test suite from that repository still passes with the Copilot-suggested alternative source code. (A weaker version would check that linters agree with the code.) The idea is to use a particularity of code: there are (often) unit tests that you can execute, and the code comes with its own functional test already present. On the other hand, in some domains, you might not be able to construct any functional tests you can execute programmatically. But there is one last arrow a prompt engineer such as yourself always has in their quiver: the model itself.

LLM assessment

The quality of a natural language response to a problem is often a wooly, hard-to-pin-down affair. If the LLM outputs a number, you can easily compare it with the gold standard; if the LLM outputs a classification, you can directly compare strings to determine accuracy; and if the LLM outputs a program, you can run unit tests.

But if the LLM outputs a textual answer to a question, how do you measure how *friendly* and how *helpful the response is*? Fortunately, such evaluations are where LLMs shine, and you can use them to assess the response. On the other hand, maybe this isn't a good idea—after all, it was probably the very same LLM that produced the response that we're evaluating, and now, you're asking it to grade its own work to see how good it is. Isn't that a bit like giving a high schooler an assignment to write an essay and then asking them to grade their own work? The answer is *no*—at least not if you do it right.

Even though the questions to the LLM are often formulated as absolute quality questions (e.g., "Is this correct?"), an LLM assessment a priori serves only as a relative quality judgement (e.g., "Version A is considered right more often than version B."). You may get assessments like "The LLM judges the application to be correct in 81% of cases," and on their own, these carry little meaning.

If you want to correctly use an LLM to assess its own work, then you shouldn't let the LLM think it's grading its own work. Assessments are a kind of advice conversation, and as you already know from Chapter 6, advice conversations work best when the model thinks it's grading a third party. In fact, while models get a bit less accurate when they think they're being asked to grade the user compared with when they think they're being asked to grade a third party, they usually get much worse when

they think they're being asked to grade themselves, because they're suddenly subject to a host of conflicting biases. Most models' training data includes a good chunk of forum discussions (or even comments), which aren't exactly known for objective self-reflection. On the other hand, if the model is subject to RLHF, then to please its human evaluators, it often learns to veer toward the other extreme, falling over itself to correct its output on even the slightest expression of user doubt. Even if a model manages to strike a balance on average, being pulled in different directions isn't conducive to it providing an objective analysis.

SOMA Assessment

Another good way to optimize the LLM for assessment is to try to use what we'll call a *SOMA assessment*, which consists of specific questions (S), ordinal scaled answers (O), and multiaspect coverage (MA). Let's talk about each of the parts of SOMA assessment.

Specific questions

There are tasks in which verifying a solution is much easier than coming up with one. For example, it's hard to invent a limerick on the spot, but it's easy to confirm whether a given poem fulfills the criteria for being a limerick. If your application task happens to be one of them, you might get away with asking "Is this right?" But in most cases, there's little information gained from a generic assessment like that. In Figure 10-3, we had the example of a smart home system reacting to the user voicing "I'm a little chilly," with `to=functions.set_room_temp {"temp": 77}`. Answering the question, "Is the completion right?" is not much easier than coming up with the completion in the first place. In fact, the answer to the assessment might well be worse than the original generation, because there are several ways to interpret it.

Ordinal scaled answers

One such ambiguity is that it's unclear how good a completion would need to be to be "right." It's no good if the standards that an individual answer is held to depend on the model's capriciousness and the next answer is held to a different standard. It's even worse if, instead of a random effect, there's a systematic bias, such as the model holding answers trying for more accuracy to higher standards or accepting generally OK answers (that are more than 50% correct) while rejecting almost perfect answers (that are not *completely* right).

The solution is to ditch yes/no answers in the first place and ask the model to rate the completion on an ordinal scale, where it's not only easier to convey nuance but also to obtain consistent measurements by communicating the meaning of these numbers.

For example, if you ask the model to rate on a scale from 1 to 5,[1] you can add a description or examples for each of these levels, as shown in Example 10-1.

Multi-aspect coverage

But "How good is good?" is far from the only source of ambiguity in getting an answer to a question like "Is the completion right?" If you think of different completions for "I'm a little chilly," the model might focus sometimes on the question of whether the suggested room temperature is correct, sometimes on the question of whether the assistant should ask before changing the temperature, and sometimes on whether `set_room_temp` is the right function to use. Such inconsistencies are pretty bad if you want to use the model assessment in a systematic way.

The remedy here is to control these multiple aspects explicitly: instead of asking the model how good a suggestion is and praying it's always using the same criterion to judge goodness, you can prepare in advance a couple of categories to judge the model on and ask the model to rate the suggestion in each category. For the smart home assistant above, the categories could be as follows:

- Whether the completion succeeded in implementing the action the model intended (making the correct choice of tool called with correct syntax)
- Whether that action remedied the user's problem at hand (being chilly)
- Whether the model was sufficiently restrained from doing something crazy without asking and sufficiently assertive to not need too much hand-holding

Then, instead of asking one question, you're asking three, and you're either adding up the scores or looking for more complex patterns.

Remember to spell out the fact that you're doing an assessment and which aspects to grade on, before showing the example to the model. After all, the LLM can't backtrack and can read through the text only once. If the question precedes the example to evaluate, then when the LLM reads through the example, it does so with the evaluation framework already in mind, and it can focus on the right aspects.

When you're choosing these aspects for grading an application, it's important to choose the right ones. One common approach is to focus on the aspects of intent and execution:

1 Research from psychometrics indicates that 5 is a pretty good default, in fact.

- Did the model have the right intent? For example, is turning up the heat to 77°F really the solution to the user's problem?
- Did the model correctly execute upon that intent? For example, did the model use the correct tools and tool-calling syntax?

For example, you can ask chat applications offering advice to the user whether the advice addressed the right things. If the user asked for things not to miss when visiting Morocco, you can ask the app whether it actually provided the intended sightseeing information (rather than telling the user not to miss their flight) in a complete way (rather than only listing the best cafes). Also, you can ask the application whether the advice was actually correct. These aspects form the basis for the relevance-truth-completeness (RTC)[2] system that was originally developed for scoring chat conversations by GitHub Copilot.

You should break apart any Goldilocks questions that ask whether a completion was "just right." Those questions really capture two aspects: it was enough, and it wasn't too much. You typically get cleaner results if you ask these questions separately.

SOMA mastery

Taking it all together, a SOMA assessment asks specific questions on an ordinal scale covering multiple aspects, like in Example 10-1. SOMA acts like a guardrail by defining the evaluation task so precisely that the model has no choice but to be objective in its assessments…at least we hope. But how can you be sure that it works? How do you choose your questions, aspects, and descriptions of the ordinal options correctly, and how can you be sure they didn't just go over the model's head?

Example 10-1. Asking the LLM to rate one of the chosen aspects

```
I need your help with evaluating a smart home assistant. I'm going to give you
some interactions of that assistant, which you are to grade on a scale of 1 to 5.
Grade each interaction for effectiveness: whether the assistant's attempted action
would have remedied the user's problem.

Please rate effectiveness on a scale of 1 to 5, where the values mean the
following:

1. This action would do nothing to address the user's problem or might even
make it worse.
2. This action might address a small part of the problem but leave the main
part unaddressed.
```

2 Lizzie Redford, "Machine Psychometrics: Design & Validation Principles for LLM Self-Evaluation"

```
3. This action has a good chance of addressing a substantial part of the problem.
4. This action is not guaranteed to work completely, but it should solve most of
the problem.
5. This action will definitely solve the problem completely.

The conversation was as follows:

User: I'm a bit chilly.
Assistant: to functions.set_room_temp {“temp”: 77}

Please provide a thorough analysis and then conclude your answer with
"Effectiveness: X," where X is your chosen effectiveness
rating from 1 to 5.
```

The answer is that you should ground your model evaluation in human evaluation. The good thing about the model is that it scales, while people don't (Elastigirl notwithstanding). Therefore, using LLMs to assess their own performance is basically a replacement for using human annotators, and you want to make sure that you suffer no substantial regression by doing that. You could let a human annotate some cases and compare, but all you'll find out is that there's some amount of disagreement between the human and the model—and that's normal. Humans disagree, too, so what you actually need to do is let *several* humans answer the questions. Then, you need to confirm that the disagreement among this pool of human assessors (measured through some standard method like Kendall's Tau) remains stable if you add the model (queried once, at temperature 0) to the pool.

The following lists summarize the offline evaluation choices. Note that for offline evaluation, you need a source for inputs and a test for outputs. These lists include the main kinds with what we consider the most critical question; if you can't find a way to answer with yes, then you can't use that row.

Pick one source:

Existing records
: Can you find plenty of them?

App usage
: Is the data trickle fast enough (also considering old data invalidation through app changes)?

Synthetic examples
: Are you willing to spend the time crafting the synthesis procedure?

Pick one test:

Ground truth match
: Is a (complete or partial) match realistic and meaningful?

Functional test
: Can you isolate a critical aspect that can be automatedly assessed?

LLM assessment
: Are good and bad outputs recognizably different (by people, say)?

Online Evaluation

All of the methods in the previous section are at least a bit artificial. They test the model's performance in the lab, not in real life. There are three advantages of assessing your app in the lab:

- The lab is safe, and if you mess up in there, no one will know.
- The lab scales much better, so you can try out more ideas more quickly.
- The lab exists before your app ever ships, so you can start assessing earlier.

But, as Opus told us in their famous song, "Life is live," and it's hard to beat that. If you run an app in real life, then you've got actual users in the loop, and application performance with users is the ultimate test of whether the application has true merit.

A/B Testing

The standard way to learn from users is through *A/B testing*: you ship two (or another small number) of alternatives—let's call them A and B—to see which performs best. Normally, one of those alternatives will be the status quo, and the other will be your modification you want to assess. Hopefully, you've already performed offline evaluation of the alternatives, to whittle down the number of possibilities to test and also to avoid putting some real stinkers in front of your users. You define in advance which metrics to look for that you want to optimize; often, they are proxies for user satisfaction (e.g., average rating, acceptance rate). You may also define some guardrail metrics that you don't want to increase; often, they are proxies for catastrophic failures (e.g., errors, complaints). Then, a random selection of users gets the app running in mode A, and the rest gets the app running in mode B. You let the experiment run for a while, collect the metrics you've decided on, and see whether A or B is better. Then, you roll out the winning alternative to all users.

Online evaluation typically has less bandwidth than offline evaluation. You have only a finite number of users, and getting a signal can take some time, so be deliberate in which ideas you test online.

A/B tests aren't unique to LLM applications, and there are plenty of established solutions for taking care of assigning experiment groups to users or sessions as well as taking care of statistical analysis. These solutions include Optimizely, VWO, and AB Tasty, and they all rely on your app being able to run in two modes: alternative A and alternative B. For example, if A is your current prompt-engineering logic and B is a new prompting idea you want to try out, then you need your app to be able to perform either, depending on some flag being set by the A/B testing setup. If your app runs client-side, that means that you need to roll out an update with the new prompting idea to all (most[3]) users before you can even start testing it. That rollout time introduces another significant reason why A/B experimentation often moves slower than offline evaluation.

To achieve successful online evaluation, your most important initial goal must be to determine which metric(s) you want to optimize for. That's what determines how you decide which alternative is "better." Let's review an earlier example about an application that suggests travel destinations. A user in group A gets the suggestion "Monaco," and a user in group B gets the suggestion "Chicago." What signal should you listen for to be able to say whether those were great suggestions or bad ones? To answer this question, let's get an overview of the possible metrics.

Metrics

There are five main kinds of metrics. From most to least straightforward, they are as follows:

1. Direct feedback: what does the user say to the suggestion?
2. Functional correctness: does the suggestion work?
3. User acceptance: does the user follow the suggestion?
4. Achieved impact: how much does the user benefit?
5. Incidental metrics: what are the measurements "around" the suggestion?

We'll explain them all in turn, starting with direct feedback (see Figure 10-4).

3 You can't just say, "Let's put the users who already updated in group B ('the new thing') and the others in group A ('the status quo')," because users who update quickly usually behave differently from users who update less regularly. What you can do is say, "Let's test only users who have already updated, and those who haven't updated are members of neither group A or nor group B; they simply don't take part in the analysis."

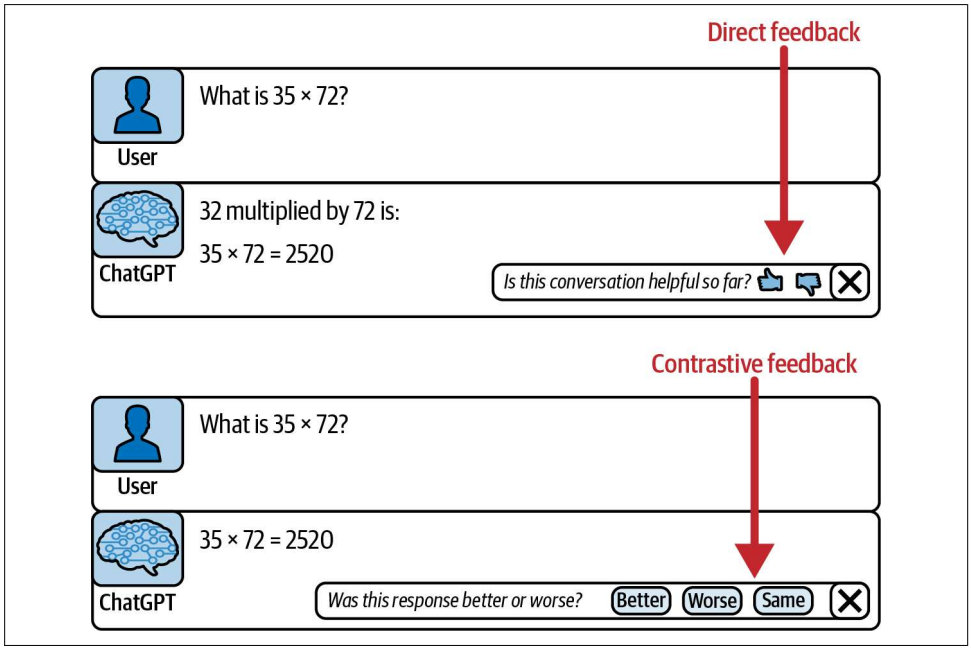

Figure 10-4. Two different ways in which ChatGPT elicits direct feedback

Have you noticed how regularly ChatGPT asks its users for their opinion? It makes sense: conversations are hard to evaluate, and who better than the user to assess them? A small thumbs-up or thumbs-down button next to answers gives the user a quick way to voice their delight—or an outlet for their frustration. The latter is more common, and typically, it's also a reliable signal: thumbs-ups, if optional, are normally not given for solid performance, but only for particular brilliance, and that dilutes the signal. (Maybe that's why OpenAI stopped displaying thumbs-up buttons for single conversation items.)

ChatGPT occasionally goes one step further than classical A/B testing toward *contrastive A/B testing*, asking, "Which one of these two suggestions is better?" That leads to possibly the clearest signal, but it's also extra intrusive—and all requests for direct feedback are pretty intrusive already. If your app is an assistant (like ChatGPT) that people seek out and communicate with very deliberately, you can get away with that, but no one wants their smart home system to constantly ask, "And how was it for you?" each time it adjusts one of the lights.

In many applications, feedback is more valuable if it's delayed—it's one thing for the user to appreciate the vacation suggestion of Chicago and be skeptical of the idea of going to Monaco, but if you're aiming to provide proper value to the user, it's even more valuable to learn, after the fact, that the suggested trip to Chicago kicked ass and the trip to Monaco sucked.

The data you gather using direct feedback is usually very high quality—in addition to using it for evaluation, you can use it as training data for model fine-tuning.

Metrics testing for *functional correctness* emphasizes the more objective parts of an LLM application: the app tried to do something, but did it work? Sometimes, you can easily check at least a partial aspect: the code compiles—and that's good (although the code might not actually do the right thing); you get a ticket confirmation—and that's good (although you might not have booked the right destination). At other times, signals for functional correctness are much more concrete and certain, especially for smaller subtasks in a greater routine. You wanted to open a program, but is it running? You wanted to send an email, but is it in your outbox?

If you can't assess the suggestion directly, most applications can check whether the *user accepts* them or at least takes steps to accept them—for example, did the user end up booking a trip to Chicago? Sometimes, that's as direct as click-through rate: if your suggestion contains a link, how often do users click on it? That affirms only that a suggestion looked promising, not that it was actually useful, but quite often, a reasonable first start is actually the most important thing.

That turned out to be our finding for Copilot, when we found that acceptance metrics correlated more strongly with the productivity gains reported by the user than with more sophisticated *measurements of impact*. Those are related signals that try to assess the same question: did the user find the suggestion helpful? But they come from the other direction by looking at the final outcome. Here, you find metrics like "In the end, how much of the email was written by the assistant?" or "When the user clicked on the suggested travel destination, did they actually end up buying a ticket?"

Finally, each application comes with a gaggle of *incidental metrics* that measure relevant aspects, but not necessarily with a unique relationship to "goodness." In interactive scenarios, the most important one of these will be latency, even though a lightning-fast suggestion can still be worthless and a more measured one can still be worthwhile. Conversational assistants typically also track conversation time, even though it's far from clear in general whether a short conversation is good (i.e., the problem is solved instantly, and the user is completely satisfied) or bad (i.e., the assistant showed its incompetence from the start, user rage-quits). Typically, it's better to track more incidental metrics than fewer, both as rough indicators of quality (e.g., you might have some idea that long conversations are often better) and to prompt an investigation into any unexpected changes.

And there you have it: lots of different ideas to choose from. It's worth spending some time investigating which kinds of metrics you can collect for your use case and how confident you are in their value. The most likely case, and where you should start looking first, will be an acceptance or impact metric. If you don't find one that you can be confident of, you'll have to ask for direct feedback. But even then, you'll probably keep some acceptance or impact metrics as guardrails, monitoring that they don't regress, and you'll likely keep some functional correctness and incidental metrics (particularly latency and errors) as well.

Conclusion

Evaluation is an important subject, but it's difficult, due to the many knobs you can twiddle. Does your offline evaluation get its examples through existing records and historical app usage, or does it make them up synthetically? Do you test them by comparing them with a gold standard, do you automatically check their functionality, or do you assess them using the LLM itself? Does your online evaluation track user feedback, functional correctness, acceptance rates, or impact? And which incidental metrics do you add?

The perfect choice is different for each application. But what is always true is that evaluation is essential for the continued development of your app, and any time spent on this area is time well spent.