O'REILLY®

Compliments of

Observability for Large Language Models

Understanding and Improving Your Use of LLMs

Phillip Carter

REPORT

Observability for Large Language Models

Understanding and Improving Your Use of LLMs

Phillip Carter

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

Observability for Large Language Models
by Phillip Carter

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: John Devins
Development Editor: Corbin Collins
Production Editor: Elizabeth Faerm
Copyeditor: Paula L. Fleming
Proofreader: Rebecca Gordon
Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Kate Dullea

October 2023: First Edition

Revision History for the First Edition
2023-09-28: First Release

This work is part of a collaboration between O'Reilly and Honeycomb. See our statement of editorial independence.

978-1-098-16598-7

[LSI]

Table of Contents

Observability for Large Language Models

Artificial intelligence (AI) has revolutionized numerous industries, enabling organizations to accomplish tasks and solve complex problems with unprecedented efficiency. In particular, large language models (LLMs) have emerged as powerful tools, demonstrating exceptional language-processing capabilities and fueling a surge in their adoption across a wide range of applications. From chatbots and language translation to content generation and data analysis, LLMs are being adopted by companies of all sizes and across all industries.

As organizations eagerly embrace the potential of LLMs, the need to understand their behavior in production and use that understanding to improve development with them has become apparent. While the initial excitement surrounding LLMs often centers on accessing their remarkable capabilities with only a small up-front investment, it is crucial to acknowledge the significant problems that can arise after their initial implementation into a product. By introducing open-ended inputs in a product, organizations expose themselves to user behavior they've likely never seen before (and cannot possibly predict). LLMs are *nondeterministic*, meaning that the same inputs don't always yield the same outputs, yet end users generally expect a degree of predictability in outputs. Organizations that lack good tools and data to understand systems in production may find themselves ill-prepared to tackle the challenges posed by a feature that uses LLMs.

One answer to solving these problems lies in software observability. *Observability* refers to the ability to understand an application's behavior based on the data that it generates at runtime, called *telemetry*. The rise of observability in modern software comes from a need to understand the constantly changing, often nondeterministic behavior of applications. The nature of interconnected cloud services with ever-changing infrastructure and application code across several teams makes traditional debugging impossible, so unreliability increases significantly without a different set of tools. As it turns out, the problems posed by modern software systems are very similar to those of LLMs. As such, the tools and practices of observability are a good fit for taming the complexity and unreliability of product features using LLMs.

This technical report dives into the realm of observability for LLMs. It covers why observability matters for modern AI and proposes a methodology for tracking a small but valuable amount of data in production and then using that data to drive product development and prompt engineering practices. Finally, it elaborates on the roles and responsibilities of different teams in a world where product features undergo an *observability loop* of monitoring telemetry, using snapshots of that telemetry to improve features, monitoring telemetry from those improvements, and so on.

The details and guidance in this report are from the real-world experiences of several organizations that applied observability practices to their products that use LLMs. In particular, several instances are drawn directly from Honeycomb's product Query Assistant, showing how Honeycomb was able to apply these practices to bring the feature out of experimentation and turn it into a core product offering.

Why Observability Matters for LLMs

LLMs represent a step change in the capability and accessibility of machine learning (ML) models for organizations. Every product has problems to solve for its users where there is no single solution but rather a set of solutions lying on some spectrum of "correct" or "right." Traditionally, companies turned to AI to solve these problems, but at great cost. Now, much of that cost has evaporated, and any product engineering team—even if they have no experience with ML—can solve problems using LLMs through a simple API.

However, the very things that make LLMs so useful also give rise to the biggest challenges. End users expect powerful capabilities with reliable behavior, but steering an LLM to reliability for all possible inputs is challenging. Furthermore, the tools that product engineers traditionally lean on for improving reliability—step-by-step debugging and unit testing—aren't feasible with LLMs.

LLMs introduce reliability and predictability challenges that can seem scary when released to production. They are black boxes (you can't debug them like you can a single-threaded client application) that produce nondeterministic outputs based on natural language inputs. Let's unpack that a bit.

Natural language inputs are broad. Very broad. A natural language, such as English, is infinitely more expressive than any programming language, query language, or UI. What this means is that users of applications with natural language inputs will do things you cannot hope to predict. Yes, there will likely be patterns of similar inputs that users will input for specific reasons—you can account for those—but there is an extremely long tail of inputs your users will create, and users expect those inputs to be handled well.

On the other end of the spectrum, in many cases an LLM is completely un-debuggable in the traditional sense. You can't load debug symbols and step through each phase of execution with an LLM like you can with most software. Unless you're an ML researcher by profession, chances are you don't have any way to explain why an LLM emits a particular output for a given input. Yes, there are some controls you can place on it to affect the randomness of outputs—namely, the *temperature* and *top_p sampling* parameters—but they don't guarantee repeatability of outputs. And when you combine this with an excessively broad class of inputs like natural language, you're left with a system that's incredibly powerful but very difficult to understand and guide into good behavior.

Unit and integration testing, which involves verifying that specific inputs yield specific outputs, falls short when applied to LLMs. The range of possible outputs is vast, making it impossible to exhaustively test all potential inputs and scenarios. Instead, ML teams often build what are called *evaluation systems* that can evaluate the effectiveness of a model (or a prompt, in the case of most uses of LLMs). These evaluation systems are powerful and, over time, play a role similar to that of unit testing for most software systems. However,

to get an effective evaluation system bootstrapped in the first place, you need quality data based on real use of an ML model. So there isn't a drop-in replacement for unit tests that you can just start with.

Finally, attempting to de-risk a product launch through early access programs or limited user testing can introduce bias and create a false sense of security. Early access programs and user testing often fail to capture the full range of user behavior and potential edge cases that arise in real-world usage with a wide range of users. More importantly, when gauging ambiguous criteria, like user experience (UX), they suffer greatly from selection bias and the biases introduced by asking users to use the product in particular ways. Effectively working around these problems requires significant time and money. Couple this with widespread organizational tendencies to always report successful results from expensive time investments, and you'll be left with a sense of security that comes crashing down when you truly go live. Instead, it's better to embrace a "ship to learn" mentality and release features earlier, but you need a way to systematically "learn" from what was shipped.

Put differently, you should be aware of the following things when building with LLMs:

- Failure will happen—it's a question of *when*, not *if*.
- Users will do things you can't possibly predict.
- You will ship a bug fix that breaks something else.
- You can't write unit tests for LLMs (or practice test-driven development).
- Early access programs won't really help you.

What's interesting is that these properties aren't unique to LLMs, but LLMs make these problems more apparent. Modern applications that deal with orders of magnitude more complex than they did a decade ago all exhibit these same behaviors. To deal with that reality, engineering teams have turned to observability as a better way to debug, monitor, and use data from production to inform product improvements. By collecting relevant information about their applications from within their application code and systematically analyzing and monitoring this data, teams can wrangle modern systems. This principle can apply equally to products that use modern AI systems.

As systems get more complex, with nondeterministic outputs and emergent properties, the only way to understand them is by instrumenting the code and observing how users use it as it runs in production. LLMs are on the far side of a spectrum that has been elongating and is becoming ever more unpredictable and unknowable for years. Observability, as a practice and as a set of tools, tames that complexity and allows you to systematically understand and improve your applications.

A Quick Primer on Prompt Engineering

Prompt engineering (also called *prompting*) is a set of methods for telling an LLM to steer its outputs toward a desired result without changing the model. These methods of communication are primarily one or more textual inputs that may contain instructions, data, user inputs, example outputs, and more. By using prompt engineering rather than training a model from scratch or fine-tuning an existing model, you can achieve a wide range of behaviors through a simple AI call.

For example, let's say you want to use an LLM to produce a SQL query based on natural language input. A prompt for this task might contain the following text, with several placeholders that parameterize relevant data for the task:

```
You are an AI that turns natural language input into SQL queries.

The table you are querying is: <table_name>
The columns in this table are: <column_list>

Given user input, the table, and its columns, produce a SQL
statement.

Input: Get all posts
select * from blog_posts
Input: Get posts from Alice
select * from blog_posts where author = 'Alice'

Input: <user_input>
```

Note that the preceding prompt example is illustrative only. A real prompt for translating natural language into SQL would likely be much more sophisticated, maybe even having many smaller pieces assembled programmatically such as data to parameterize based on context from a running application. Many techniques are involved in prompt engineering, and all of them have different tradeoffs that you'll need to evaluate.

The use of prompts provides a flexible approach to influencing LLM behavior and output, making it possible to tailor the generated text to specific tasks, styles, or domains. However, prompt engineering can be a subtle and nuanced process. Even slight modifications to the prompts can yield dramatic differences in the outputs produced by the model. The choice of wording, phrasing, or context within the prompt can significantly impact the generated responses. Prompt engineering requires a rigorous approach to experimentation and interaction, which is covered in more detail later in this report.

A Quick Primer on OpenTelemetry

To gather valuable data about user behavior and system performance of LLM-based products, it is crucial to instrument the underlying code effectively so it produces relevant telemetry that can be analyzed later. This is where OpenTelemetry comes into play. *OpenTelemetry* is an open standard for creating telemetry, providing a common framework for capturing and collecting observability data.

One of the major data types that OpenTelemetry supports is a trace. A *trace* (also known as a *distributed trace*) is a collection of *structured logs*, known as *spans*, correlated by an ID and given a duration. Additionally, spans can identify a parent span, allowing you to represent a hierarchy of operations in data. By using traces, you can quantify the "path" that a request makes to a system, gathering relevant information at any point along the way. You can break up this "path" into relevant suboperations with their own durations, as illustrated in Figure 1.

Although traces are designed to support collecting data across arbitrarily distributed systems, they are equally useful for simple transactions in a single application. By using traces with OpenTelemetry, you can track information about user inputs, AI model outputs, and the result of any operations you perform on AI model outputs

before showing a result to an end user, and you can do so using a common standard that countless tools support.

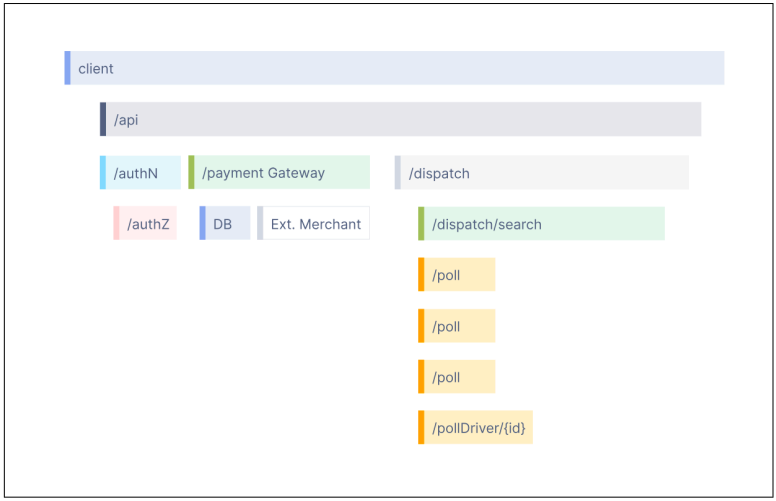

Figure 1. An abstract example of a trace that captures how a request moves through an application

To use OpenTelemetry for modern AI observability, you need to do a few things:

- Install automatic instrumentation or a relevant instrumentation library to track incoming and outgoing requests. This lets you track the behavior of external API calls, such as calls to OpenAI, and correlate that information with a user request to your own apps.
- Install the relevant OpenTelemetry software development kit (SDK) for your language into your codebase and use the OpenTelemetry APIs to create manual instrumentation that captures relevant data and operations before and after a call to a generative AI model.

By combining OpenTelemetry's automatic tracing instrumentation capabilities with manual instrumentation, you can capture everything you need to begin systematically analyzing user behavior. In this way, you can learn how user behavior impacts the results that a generative AI model will produce.

Designing Your Telemetry

To track everything you need using OpenTelemetry, you'll first need to ensure you have an SDK initialized in your codebase and have requests and responses instrumented, either via an instrumentation library or automatic instrumentation.

The telemetry you'll need to gather depends on how you interact with an LLM. The following broad categories provide a guideline for designing your telemetry.

Simple LLM Call with a Static Prompt

Some prompts are mostly static. That is, they might include user input and some additional info, but inputs are largely the same each time. For these cases, you just need three spans:

- The overall tracking span for all operations. This is important because it's the span you'll usually query for later on. It represents the end-to-end user experience of your feature that uses an LLM.
- A child span of the overall tracking span that tracks the call to an LLM (this can be done with automatic instrumentation).
- A child span of the overall tracking span that tracks any parsing or validation of outputs you do on LLM outputs.

For many applications, these three spans will suffice for your observability needs.

Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is an extremely common pattern for applications using LLMs. The idea is that you store vector embeddings of a knowledge base or other data, create a vector embedding for a user's input, use matrix math to pull out the subset of the knowledge base that's relevant to the user's input, and pass only that subset to an LLM. RAG is an essential part of many products or features that use LLMs. One of the most common use cases is as follows:

1. Calculate embedding vectors for pieces of data in your application, either up front or as a cron job.
2. Store those embeddings somewhere, either in a vector database, a Redis cache, or an in-memory index of some kind (such as FAISS).
3. When a user makes a request, calculate an embedding of their input.
4. Use a similarity function, such as cosine similarity, to calculate which pieces of data are the most relevant to the user's input.
5. Include only the relevant subset of data into a prompt rather than the full data (which may be too large to fit into an LLM API call).

You'll need more spans in your telemetry to track what's going on when using embeddings:

- The overall tracking span for all operations. This is important because it's the span you'll usually query for later on. It represents the end-to-end user experience of your feature that uses an LLM.
- A child span of the overall tracking span that represents the call to calculate an embedding vector for user input.
- A call to your system that stores vector embeddings that selects a relevant subset of data based on the user input vector embedding. Critically, it should include some information about which subset was chosen, for example, the names of each document selected from a knowledge base.
- A child span of the overall tracking span that tracks the call to an LLM (this can be done with automatic instrumentation).
- A child span of the overall tracking span that tracks any parsing or validation of outputs you do on LLM outputs.

Additionally, you may wish to monitor a cron job that calculates and updates embeddings for the data you're working with. This is likely a separate trace in a separate system, but if it's worth monitoring to you, then you'll need at least one span that tracks calls to the embedding model.

Agents or Chained LLM Calls

Several use cases for LLMs involve chaining calls or agents. In these cases, the telemetry you design isn't fundamentally different from the other use cases. However, you'll want to ensure that you're doing two additional things:

- A true overall tracking span tracks every step or iteration of an agent or LLM call chain.
- Each span that deals with LLM inputs and outputs is named according to each step or iteration, and it correctly logs the specific inputs and outputs for that step.

Instrumenting an agent or LLM chain isn't difficult, but be wary of unbounded execution! When an agent or chain can take a varying number of steps, your traces don't have a fixed number of spans.

Capturing Errors

OpenTelemetry captures errors from automatic instrumentation, but it doesn't automatically capture errors from your own application. To do that, you'll also need to handle your own errors and attach them to the relevant span they occur within. This is a standard practice for any manual instrumentation process with OpenTelemetry, and your language SDK will provide the necessary APIs to do this.

Dynamic Prompt Building

If the act of assembling user input and a prompt is an involved process, you may also want to track that operation as a separate span. Although it's not usually a time-consuming process to build a span dynamically, you may have complex logic that determines how pieces of a prompt are assembled at runtime, and spans are how you capture information about that complex logic. The criteria for this decision-making process should be captured in a span, which then lets you isolate any interesting behavior based on how a prompt is generated.

Cost Is Important but Usually Not Critical

If you're using an external service, such as OpenAI, you pay for each API call you make. Because of that, you'll want to monitor your cost. OpenAI and other vendors offer a way to track usage on a daily or monthly basis, which is enough for most organizations. However, if you desire or need more precise calculations, you can track cost at a more granular level. The way to do this is by counting tokens and keeping track of your total cost on a dashboard.

For LLMs, *tokens* are an encoded representation of input and output text. When you pass input text to an LLM, that text is encoded into a list of tokens that represent the text in an efficient way. When the LLM responds, it does so by emitting one token at a time, which is then decoded before a response is sent (or stream-decoded when a response is streamed). Most vendors, like OpenAI, charge based on the number of tokens passed as input and the number of tokens output by the LLM you use. Therefore, you can easily calculate your total cost based on these token counts and the price per token.

However, it's important to know that cost is rarely a primary concern. Most LLMs are quite inexpensive to use today, and they'll likely just get cheaper over time as they become more efficient and face external pressures from competition. Additionally, to protect their infrastructure, vendors apply rate limiting, and these rate limits place boundaries on usage that are extremely economical. Your cloud vendor bill is likely to be significantly higher than your LLM vendor bill. You can also apply rate limiting to your own application that uses an LLM, which, when combined with easily calculated cost and vendor rate limits, can make estimating your monthly bill very easy.

It's All About Inputs and Outputs

Because LLMs and generative AI are nondeterministic black boxes with unbounded variations on inputs they can receive, getting good observability into these systems is all about systematically tracking inputs and outputs. Specifically, there are five major things to track:

- User inputs
- LLM output
- Value(s) of data after parsing/validation of LLM outputs, if there is no error
- Any error, whether from LLM output or parsing/validation of the LLM output
- User feedback, such as thumbs up/down responses, if relevant

This assumes you have a way to track user feedback in your telemetry. It's not a requirement, but it does help your observability and prompt-engineering efforts significantly, so you should invest in it if you can.

Why You Should Parse and/or Validate LLM Outputs

As a brief aside, it's critically important to parse and/or validate LLM outputs if you can. For most applications, you'll want an LLM to output something in a particular structure or output pieces of data that you can parse out and assemble yourself afterward. There are several reasons to do this.

First and foremost, LLM outputs should be considered *untrusted inputs to your system*. In particular, *prompt injection attacks* are a common way for malicious actors to try to exfiltrate data, manipulate outputs for other users, or reprogram a part of your application to do all kinds of bad things. Although there is no complete solution for prompt injection attacks, there are ways to mitigate their ability to impact your systems. One such way is to parse output of an LLM into a particular kind of structure that you can validate before you pass it to another component in your application or show to users.

Secondly, forcing an LLM to output in a particular way and then parsing that output allows you to use LLMs for all kinds of applications, not just basic chatbots. Parsing these outputs allows you to validate them against a set of rules, which is critical to using those outputs in another part of an application or displaying them to users.

Additionally, there are prompt-engineering techniques that involve having an LLM output pieces of an answer that you manually assemble into the complete answer later. This is often advantageous because it reduces the complexity of a task for the LLM. It also

allows the LLM to be more accurate due to not having to keep track of the relationships between the pieces of text it generates. If you use a technique like this, you must parse outputs from the response.

For example, with Honeycomb's Query Assistant feature, LLMs are used to convert natural language into a Honeycomb query object. This object follows a particular structure; pieces of the object have particular rules for their structure—otherwise, the query object is considered invalid. The LLM produces these objects, which are then later parsed and validated. One of the major benefits of this parsing and validation is that when it does fail, it can produce specific—and often correctable—errors, allowing for a set of "fixups" to the data returned by the LLM. This programmatic correcting of LLM outputs can actually yield impressive results, which I'll elaborate on later.

Finally, not all outputs can be parsed into a structured form that lends itself well to validation through a set of rules. The accuracy of an output may depend exclusively on its interpretation by an end user. There is less direct benefit to parsing and validating responses in these cases.

Why You Should Capture Both LLM Outputs and Final Outputs

When you parse and validate LLM outputs, the final output (assuming there's no error) is often in a different format from what the LLM initially responds with. Furthermore, especially when combined with other data, the result that a user sees may have no resemblance to what the LLM produced. Not only is this fine, but it's often expected for a robust feature that uses LLMs. However, it means you need to capture the data the LLM outputs and your final outputs in your telemetry.

You need the LLM output because you need to know how an LLM responds to your prompt that includes user input. Without this information, it's impossible to improve your prompt through prompt engineering. This holds true whether you are using a fine-tuned model or a foundational model directly. Additionally, as I'll elaborate later, this data is essential for creating an accurate evaluation system that lets development teams rapidly iterate on prompts with reasonable assurance that they won't regress behavior when they deploy new changes.

While the LLM output provides insights into the model's response, it is equally important to track the actual output that is presented (in some form) to a user after any parsing or validation steps. If you have a user feedback system, it is these outputs that users will judge. Tracking what's closest to what they interact with is crucial to understanding the entire journey of a request that involves LLMs.

Why You Should Track All Errors

Tracking errors—whether they arise from a network error, a timeout, the LLM itself, or the parsing and validation process—is crucial for understanding what kinds of user inputs can lead to errors. It practically goes without saying, but a feature that causes users to experience errors all the time is a bad feature. The only way you can know which errors users experience the most is to track all possible errors related to a call to an LLM and what you do with its outputs. Additionally, as you'll see later, you can often correct an LLM output directly if it fails a parsing/validation step—but you won't know how often that occurs unless you track the error associated with that parsing/validation step in your telemetry.

Acting on Inputs, Outputs, and Errors

Now that you finally have user inputs, LLM outputs, validation/parsing outputs, and errors, you can start analyzing them!

First, using an observability tool, write a query that groups requests by errors and by frequency. As you'd expect, this query will show which errors are the most common, and you can use the output as a prioritized list of which issues to fix first.

However, observability really shines when you can group by many different data points. Instead of just looking at errors, you'll want the whole picture: an error, the input that the user gave, the output of the LLM, and, if it exists, the output of parsing/validation. With a multidimensional grouping like this, you'll be able to start answering the following questions and more:

- Are users repeatedly entering the same inputs when they get an error or something they don't like?
- What commonalities exist in user inputs for the same error?

- Are there any similarities between LLM outputs that lead to the same parsing/validation error?
- Do identical user inputs result in different errors?

Gathering and grouping data in this way lets you pull apart and investigate all aspects of user and system behavior and feed that information back into your development lifecycle.

Tracking Latency and Errors for API Calls to LLMs

It's critical to distinguish between the latency and errors in API calls to LLMs and the entire set of operations that involve an LLM. Unfortunately, at the time of this writing, the resources that LLMs run on are in high demand, and you may find your requests rate limited heavily, timed out, or experiencing an error due to unavailable resources. These are problems that you can't control, so you should ensure you can easily separate these concerns from those that you can control. That said, there are some factors that contribute toward the latency and errors you can experience with API calls to LLMs:

- How many API calls are you making per user request? Are you generating a vector embedding for each user input prior to calling an LLM?
- How many API calls do you make per minute?
- On average, how many tokens are you passing to an LLM for each request? On average, how many tokens are you receiving per request?
- How often are your requests rate limited?
- When considering the overall latency that users experience, how much of it is due to an API call to an LLM?

These questions may not always be actionable, but they're critical to understanding the overall behavior of your product. They can also be used to evaluate LLM APIs and vendors, allowing you to quantify how well they can service your requests.

Monitoring and Service-Level Objectives

Service-level objectives (SLOs) are a way to measure data (such as requests to OpenAI) and pass it through a function (called a *service-level indicator*, or SLI) that returns true or false. You then define the rate at which the SLI should return true over a span of time. Behind the scenes, a budget for failure is established, and you can define alerts that progressively notify you when a budget gets close to—or exceeds—its limit. SLOs allow teams to act proactively on behavior and quantify improvements they make over time.

The two fundamental SLIs to track when establishing SLOs for systems involving LLMs are latency and error rates. There are more SLOs worth creating, and having an SLO that maps to an overall business objective is ideal, but latency and errors are often a good starting point.

Latency SLOs

Latency refers to the time it takes for a user to receive a result after initiating a request. In particular, it's critical to track latency for the entire lifecycle of user interactivity with a feature using LLMs. This includes the entirety of the process: the time it takes to gather input, build up or gather the prompt to an LLM, make additional API calls (such as fetching a vector embedding), make the call to an LLM, and parse/validate results.

A latency SLO is centered around a number that represents the highest acceptable latency for the entire operation. You'll most certainly want to change this number over time, but you can build an initial value by measuring how long (on average) it takes when in development.

For example, a feature that translates user input into a query language may take about 3 seconds to get a response and parse it. A good SLI would then return true for requests that take 3 seconds or less and false when latency is greater than 3 seconds.

Next, a window of time needs to be set in addition to the rate at which this SLI should succeed. As a rule of thumb, a 95% success rate over 7 days is a good latency SLO for these kinds of requests.

Here's an example: if your system makes 1,000 calls to the LLM API in a week, we expect no more than 50 of these calls (5%) to exceed the 2-second latency threshold. If more than 50 calls surpass this threshold, it indicates you're not meeting your SLO, and corrective action should be taken. Corrective action should start by looking at the inputs, outputs, and errors of requests mentioned earlier. It may be that there's nothing apparent that can be done. In that case, the best course of action is to adjust the SLO to better reflect reality in production.

Error Rate SLOs

The second SLO to monitor is error rates, which should include any error encountered during the process, whether originating from an API call or your own parsing/validation of LLM results.

An SLI for an error rate is very simple: if the overall operation contains an error, it fails. Otherwise, it succeeds.

Similarly to the latency SLO, the initial success rate will need to be established during development before releasing the feature to production. If your internal testing shows that you get a successful response 3 out of 4 times, then your success rate for your SLI should be 75%.

The error threshold should be calculated similarly to the latency SLO, aiming to meet it a certain percentage of the time over a 7-day period. For instance, out of 1,000 calls made to the LLM API, you expect at least 750 to succeed. If fewer than 750 calls are successful, you're not meeting your SLO.

The great thing about an error rate SLO is that it's one of the measures that you have the most control overt. As I elaborate later, one of the most impactful ways to improve the success rate of your feature using SLOs is to correct a "mostly correct" output from an LLM when it's considered correctable. For example, Honeycomb was able to improve the reliability rate of its Query Assistant feature from 75% to 96% in large part due to systematically tracking errors and programmatically correcting LLM outputs as they were parsed.

SLO Monitoring and Alerting

SLO alerts for LLMs should be non-urgent. They exist to *inform* so that a team can plan a corrective action rather than cause a team to halt everything and fix a problem. It's recommended to send these alerts to a messaging channel, such as one in Slack or Microsoft Teams, and never trigger an alert on platforms like PagerDuty. Unlike LLM SLO alerts, paging alerts should always be directly actionable.

When an alert is triggered, it means that the budget for failed SLIs will likely expire before the configured window of time. That's usually a sign that it's time to plan corrective action using the practices described earlier to analyze user inputs, outputs, and errors.

If no alert is ever triggered, then the SLI is probably too broad and should be adjusted. In the case of latency, maybe you need a lower latency threshold. Or perhaps your success rate should be increased for an error rate SLO. Although it's not normal to fire an alert for every configured time window, you want to iterate on your SLOs over time so that they're truly matching the representative behavior that your users are experiencing, not just "looking good" 100% of the time.

SLOs are not about striving for perfection. They exist to distill the user experience down to its most essential pieces and capture intended and representative behavior.

Feeding Observability Data Back into Development

The other half of observability involves using telemetry to inform how to improve your products that are live in production. Especially because LLMs aren't debuggable—not in the traditional sense at least—the only way you can know what's worth improving is by looking at real data from your users. When you have enough data from real-world usage, you can notice patterns where your product is falling short of user expectations—and then iterate.

But how do you do that? How much data is enough? What data should be "fed" back into development? What do you try to focus on first? How do you "unit test" your product when it's using an LLM,

which is a nondeterministic black box? This section addresses these questions.

Usage Data from Production Is Essential for Iteration

Using production data as a basis for iteration may sound like a fancy concept, but it's actually quite easy to do and requires little more than an observability tool and the ability to look at the results of a query against that data. Recall from "Acting on Inputs, Outputs, and Errors" on page 14 that you can answer many questions with the following: an error if it exists, the input that the user gave, the output of the LLM, and, if it exists, the output of parsing/validation. That's all you need to know to begin iterating!

The main thing you'll need to figure out is how much data to look at. Unfortunately, there isn't a single answer here because it depends on two factors:

- How many users are using your LLM-powered product or feature (that is, what's the volume of data)?
- The diversity of the data you have—in other words, how varied are the inputs people are passing in?

As you might imagine, if you have very little product usage or usage is primarily coming from one user, then you may not have *representative* data. That doesn't necessarily make the data bad, but it usually means you'll need to give the feature some time to accumulate enough telemetry that you have something representative.

Determining representativeness for user interactions is difficult. First of all, the data needs to be varied enough to be an accurate sampling of how users interact with your feature. You also need enough data that covers some edge cases of those different kinds of interactions. Luckily, telemetry gathered for observability has all the information you need!

As a rule of thumb, if you have more than 10 users and over 100 uses per day, then your telemetry is likely representative enough to give you confidence in the impact of prompt changes and be worth acting on. The reason you'd want even more users and unique usages is that if you have enough, you can slice your data into different categories and isolate your prompting work to address acute issues rather than overall issues. But if you can't do that at the outset, don't

worry about it. All you need to do with an observability tool is look at groupings of four things: error, user input, LLM output, and parsing/validation output. Observability tools let you group results by frequency in these fields, so you can start with one of them and look for patterns.

Intervening on Correctable Errors

In many products, LLMs are used to produce outputs that follow a particular structure rather than open-ended text. When the structure output by an LLM isn't correct, you have the option to intervene directly on that error.

Although the root of most problems in a product using LLMs has to do with prompting, you can make a surprising number of improvements without prompt engineering. Good prompt engineering is time-consuming and challenging, and if you can manually correct an LLM output without reaching for prompt engineering, it's often very effective to do so.

An example of this is Honeycomb's Query Assistant feature. When released, it produced an error of some kind about 25% of the time. This high rate of errors is quite common for LLM-powered features at the outset. When Honeycomb developers looked at some of the most common errors, they found that many were parsing/validation errors. The response that OpenAI gave back was a JSON object whose structure couldn't pass validation checks. For example, consider the following JSON object:

```
{
  "calculations":[
    {"op":"COUNT", "column":"exception.message"}
  ],
  "filters":[
    {"column":"exception.message","op":"exists"},
    {"column":"parent_name","op":"exists"}
  ],
  "breakdowns":[
    "exception.message","parent_name"
  ],
  "orders":[
    {"op":"COUNT","order":"descending"}
  ]
}
```

In the Query Assistant feature, the object attempts to represent a query for "exception count by caller," a common scenario that users use Honeycomb for. However, it's subtly incorrect. The structure that represents a `COUNT` operator is wrong because `COUNT` doesn't take a column as input. A correct response would not have included the column:

```
{
  "calculations":[
    {"op":"COUNT"}
  ],
  "filters":[
    {"column":"exception.message","op":"exists"},
    {"column":"parent_name","op":"exists"}
  ],
  "breakdowns":[
    "exception.message","parent_name"
  ],
  "orders":[
    {"op":"COUNT","order":"descending"}
  ]
}
```

In this example, the first object is mostly correct. In fact, simply removing the column field from the `COUNT` calculation object makes the query object fully valid. Moreover, this information is considered knowable at validation time because it's a validation rule. So instead of failing validation, Honeycomb decided to intervene on this kind of error by correcting the structure programmatically so it passes the validation check.

In fact, in many cases, the JSON object that an LLM responds with is mostly correct, and the information available at validation time is enough to manually correct the structure so that it can pass validation. We considered these correctable errors that didn't require further improvements to a prompt.

As a result of analyzing outputs like this and issuing only these kinds of manual corrections, the error rate in Query Assistant dropped from 25% to about 14%. Later, improvements to a prompt brought the error rate down to 3%.

Intervening on correctable errors is generally very easy. When you group four things in telemetry—error, user input, LLM output, and parsing/validation output—you'll likely find a lot of errors coming from parsing/validation checks. If you have good error messages,

you can then look at where this error is emitted and see if you can manually correct an output so that it can pass validation. When you do this systematically, you'll not only have a host of fixes, but thanks to having LLM outputs to inspect, you'll also have perfect test cases to write and verify that your corrections actually work! Although these tests won't verify that a particular user input always yields a correct final output, they will verify that a correctable LLM output can indeed be corrected.

Despite being a great source of quick wins for correctness, correcting LLM outputs manually is not sufficient for building a truly reliable and useful product. The underlying reason that correctable errors exist in the first place is a prompt that doesn't yield correct responses from the LLM in the first place. Furthermore, if you have a system with open-ended outputs with free-form text, this practice may not even apply to you. All paths eventually lead to improving your prompt and/or fine-tuning an LLM.

Building a Prompt Evaluation System Based on Production Data

There are two primary ways to improve your product's use of an LLM in the long run:

- Rigorous prompt engineering
- Fine-tuning an LLM

Realistically, even if you fine-tune an LLM, you'll still need to improve your prompt through prompt engineering. Fine-tuning an LLM is the more expensive and time-consuming process of the two, but if you require a very high degree of accuracy and reliability, it's often worth that cost. In both cases, you need some way to systematically quantify how you're improving an LLM's performance. The answer to that is through an evaluation system.

Using existing evaluation systems

Several evaluation systems based on existing large and diverse datasets are available online. For example, OpenAI's HumanEval project evaluates the effectiveness of a model for code generation tasks. If your use of an LLM is related to an existing evaluation system's focus, especially if your feature produces free-form text, then an existing evaluation system is the best place to start.

However, most existing evaluation systems aren't based on your product or data. If accuracy is specific to your domain or you're using an LLM to produce structured text that follows rules particular to your system, then these evaluations may not apply. In this case, you'll need to look into building your own evaluation system.

About evaluation systems for LLMs

An evaluation system is similar to an end-to-end suite of test cases, but it differs in a few ways:

- The system under test is a combination of a given LLM and a prompt for that LLM.
- The result of a test is probabilistic, meaning that you can't guarantee that it will produce exactly the same output each time.
- You score actual outputs relative to *preferred* outputs.
- You determine a pass/fail threshold based on aggregate scores of test runs.

The third point above is worth elaborating on. A common way to score a test run's output relative to a preferred output is through what's called a *distance function*. To do this, you need two key things:

- A set of rules that define the structure of an output, if applicable. For example, if you need an LLM to produce a JSON object with a particular structure, these rules check the validity of that JSON object structure.
- A set of rules that define what "good" output is. For example, a field in a JSON object might be a string, but a particular value in that string could be ideal compared to other values.

For each of these sets of rules, you can assign a score for how meaningful they are. For example, let's say you need an LLM to produce a JSON object. If that object is missing a critical field, then it earns a strong "bad" score. If it's missing a field that's optional, then maybe it still gets a "bad" score, but the severity isn't very high.

Using the same example, let's say the value of a field is just fine (and completely valid), but it's just not a very good value for the given input to the prompt. You can still assign a "bad" score depending on how misleading that technically correct value might be.

This process can often be involved and, critically, *requires expertise from other parts of your organization*. What makes for a "good" versus "bad" LLM output is similar to what makes for "good" versus "bad" results for an end user. "Observability Is a Team Sport" on page 26 elaborates on this.

Using LLMs to evaluate LLM outputs

For LLM outputs that are free-form text, it can sometimes be helpful to use an LLM to evaluate that output. Yes, using LLMs to evaluate LLMs is a thing people do. However, it's not an automatic solution to all your problems.

First, to effectively evaluate the output of your feature, you'll need to use a different prompt and/or LLM. This means you'll need to develop confidence in the prompt + LLM combination used for evaluations, which might mean building *another* evaluation system that evaluates your evaluator. If that sounds a little ridiculous, then hopefully it's clear by now that effectively evaluating LLM outputs is no trivial task.

Once you have confidence in your ability to use an LLM to evaluate LLM outputs, you can ask it to judge if a given output answers the user's input and contextual data. So, how do you assemble that information? By taking a sampling of data from production.

Using production data to power an evaluation system

The biggest task in an evaluation system isn't actually to build a distance function or figure out how to use an evaluator LLM. When you can base a system on good data from real-world usage, these things are quite easy to iterate on until they can be considered trustworthy.

Instead, the biggest task is building the dataset that gets evaluated in the first place. Observability is critical to creating evaluation datasets because it gathers real-world user inputs and system outputs that you can start with. You can try to create this dataset yourself first by having your own people use your product before going live. But this merely delays the inevitable. Users will use your product in

unexpected ways, and you'll want to capture those usage patterns in your evaluation system.

The most important thing about the data used to power an evaluation system is that it must be *representative*. If you don't have enough unique data points to evaluate, then all you'll gain is a false sense of confidence in your evaluations.

When you have a representative set of data from user interactions, you'll need to export the same data being used for observability purposes and the exact prompt that was used for their interaction. You'll want to then split that data by errors, resulting in two datasets: one where an interaction yielded an error and another where it did not. For each of these, you'll need to construct "preferred" answers that the LLM would *ideally* emit.

The size and quality of datasets needed for an evaluation system are a factor where there's no straightforward answer. Clearly, the more varied and unique each line of data is, the better. There's no way around this. Getting good data for evaluations is difficult and time-consuming, but it's not all or nothing, either. The more representative data you can collect over time, the better. Until then, it's better to start with a basic evaluation system than wait until you have the perfect one.

Using evaluation systems

Once you have an evaluation system with representative data, you can start to use it like any other test system. Recall that evaluation systems can assign a score to each evaluation of a prompt for a given input. This score can be systematically checked in a continuous integration (CI) system, so you then have automated testing that prevents bad prompts from going out live.

To do this, you also need to store your prompts and evaluation data in source control. As you iterate on prompts and assign versions to them, you'll want to track the performance of each prompt version. If you fine-tune an LLM, you'll want to also keep track of the version of the fine-tuned LLM and store its performance with a given prompt version. Maintaining this cross-product of prompt and LLM versions can also be done programmatically or by using a prompt evaluation tool.

If evaluation systems sound like a high-effort endeavor, that's because they are. LLMs bring great power into products, but making that power worthwhile isn't free. The more accurate you need to make an LLM for your use case, the more effort you'll need to put into evaluations. Unfortunately, the tools for good prompt engineering and evaluation systems are few and far between. There's a lot of innovation in this space, so if it seems like there aren't good tools to help you now, check back every few months.

Observability Is a Team Sport

In case it's not clear yet, effective observability for LLMs is a team sport. No single person or role in an organization can do it effectively. This is also true for any kind of software observability. It's not just something that site reliability engineering (SRE), DevOps, or platform engineers are primarily concerned with. Yes, there are certainly areas of accountability that certain roles fill, but the end-to-end operation of improving software, LLMs or not, involves many different people.

When considering products that use LLMs, several roles not typically considered as having observability enter the picture: product managers, ML engineers, and data scientists. Although you don't need to involve someone with one of those job titles to get started, you will need people who play the roles that people in these professions often play.

When considering how to improve the use of LLMs in production, it's worth thinking about the things you need to have someone in your organization understand and act upon. Someone needs to understand the following:

- How to instrument your application so it emits the telemetry you need
- How to analyze telemetry with intent to improve a feature
- How to monitor telemetry so you know your changes are working
- How to deal with end-user feedback
- How users expect to be able to use your feature that is powered by LLMs

- When your data will be representative
- How to clean and classify data effectively for an evaluation
- How to set up the necessary pipelines of production data to continuously improve evaluation systems
- How to set up developer tools and infrastructure to aid with prompt-engineering efforts, prompt lifecycle management, and systematic validation of prompt changes against an evaluation system

For small enough systems, all of the above can be done with just a few people. But the larger the product surface area and the more features using LLMs, the more people with different roles must be involved.

Some changes in responsibilities for existing roles may also be in order. Software engineers may need to concern themselves with issues of data quality, representativeness, and working with probabilistic systems. ML engineers may need to spend more time being product minded and learn about user interactions and intended product behavior. Product managers may need to brush up on Python and Jupyter Notebook so they can participate in prompt-engineering experiments. LLMs are a transformational technology, not just for products but for the roles people play in an organization.

A common theme across many organizations is that LLMs force people at all levels to understand how their users interact with their products. LLMs not only fundamentally change existing products, but they also allow entirely new categories of products and capabilities to exist. These introduce new modalities for user interaction, and it's impossible to succeed without understanding those interactions and what users expect from them.

To reiterate, you don't need to go and hire a bunch of people with different job titles now that you are using an LLM in production. But there's no free lunch for your organization. People must adapt to take on responsibilities that are not traditionally associated with the roles they were hired for—otherwise, your organization just won't be effective with LLMs in the long term.

Summary

Observability tools and practices are an essential part of modern software development. The need for them ramps up significantly when building products that use LLMs. By virtue of being non-deterministic, un-debuggable black boxes, LLMs are uniquely challenging to make reliable, and they call for a different approach to development and iteration. Teams that practice observability today will likely find that their existing tools and playbooks cross over fairly well into making LLMs in production more reliable. However, they will encounter new challenges in bringing that data back into development for core prompt engineering and/or model fine-tuning work.

In the future, you can expect innovation on the tools front:

- Automatic instrumentation for LLMs
- Better tools to manage prompt engineering lifecycles
- Better tools to pull data from production into development
- Observability tools that specialize, to an extent, on the previous points
- Better tools to make fine-tuning LLMs easier
- Turnkey evaluation frameworks to make evaluation systems easier to build and run

Although innovation will result in better tools and practices, it's unlikely that a single tool or practice will solve all the problems involved in making LLMs more reliable. Because of that, it's valuable to adopt a more general approach to making software more reliable and apply it to LLMs. Software observability is that approach.

About the Author

Phillip Carter is a Principal Product Manager at Honeycomb and leads their AI initiatives. In addition, he is a maintainer of the OpenTelemetry project, the de facto standard for observability instrumentation. In a past life, he worked on the C# and F# languages at Microsoft and helped bring the .NET stack into the modern cross-platform era.