How Distributed Tracing Uncovers Half-Truths in Your Applications
Table of contents

Summary 3
Applying the Scientific Method to Production Systems 4
The Curious Case of the Latency Spike 8
The Unreasonable Effectiveness of a Single Wide Event 15
How Tracing Uncovers Half-Truths in Slack’s CI Infrastructure 21
Conditional Distributed Tracing 26
Conclusion 32
Summary

This ebook presents leading-edge practices with distributed tracing, as described by Honeycomb customers. During Honeycomb's annual conference, o11ycon+hnycon, presenters offered a wide range of insights into how they use distributed tracing to understand complex systems and uncover elusive mysteries. While their stories come from a wide variety of contexts, many common themes emerged. In this ebook, you’ll learn about how teams at Slack, CircleCI, eero, and Intelligent Medical Objects (IMO) are using new and emerging distributed tracing practices to their advantage.

Since the advent of cloud computing, a new reality for most teams is that they have now entered an era of working with distributed systems by default. Engineering teams now cobble together a variety of different virtual infrastructure and service endpoints to create scalable, resilient, and performant applications. But rarely are they ready for what it means to now understand the intricacies of the distributed architectures they've created.

Distributed tracing is a potentially powerful tool for any team managing the operation of modern distributed software. But tracing by itself does not provide enough context to help you understand the many hidden complexities of distributed architectures. This ebook shows you how various customers pair scientific methodologies along with Honeycomb’s implementation of distributed tracing to make sense of their complex modern systems.
Applying the Scientific Method to Production Systems

Written by Pete Hodgson

Teams using Honeycomb are able to experiment quickly, with tight feedback loops that enable them to move faster. At o11ycon+hnycon, we saw that engineers find Honeycomb valuable as a way to sense and respond their way through complex environments where no one person could possibly fully understand every cause and effect. In other words, observability with Honeycomb provides a superpower: the ability to explore how your code actually behaves in the real world by applying the principles of the scientific method. Teams can form a hypothesis based on initial observations, make a small change to an environment based upon that hypothesis, and then observe again to validate or invalidate their hypothesis.

Teams that are succeeding with observability describe its adoption as a journey. It starts with a small, easy experiment, and then evolves incrementally as lessons learned are incorporated into development cycles. The theme of feedback loops and iteration continually cropped up at the conference as successful users presented their stories.

Sensing in a complex environment

Both Michael Ericksen's "The Curious Case of the Latency Spike" and Glen Mailer's "The Unreasonable Effectiveness of a Single Wide Event" presentations highlight the classic experience when observability first seems to click for many engineers—the moment they first exclaim, "My application is doing what?!" as they start using Honeycomb to analyze their systems.

That jolt of disorientation that happens when you first see what your system is really doing (as opposed to what you think it's doing) demonstrates both the challenge of understanding modern software systems and the ability of tracing and observability to shed light into these environments in powerful ways. Modern software systems have become incredibly difficult to understand due to their distributed nature. In order to make sense of them, we need processes to build up that understanding.
Dave Snowden's Cynefin model provides a useful methodology for developing that understanding. This model places systems into four categories: obvious, complicated, complex, and chaotic. It then describes how the ability to both understand and make changes to a system varies, depending on which category it is in. The Cynefin categorization of “complex” systems is an appropriate one for modern distributed software: It has moved beyond a place where understanding cause and effect only requires light analysis or expertise. Instead, the relationship between cause and effect in complex systems can only be deduced in retrospect.

In "The Curious Case of the Latency Spike," Michael eloquently evokes the feeling of what it’s like to operate within these complex systems. He described investigating production incidents like a murder mystery. Your intuition of how your own system is behaving can turn out to be dead wrong, even for those engineers who understand their systems better than anyone else.

The Cynefin model also provides guidance on how you can make progress in these complex domains. When working in a system where cause and effect are hard to understand, you should proceed by probing, sensing, and responding. In other words, you should explore the environment, inspect interesting things that you find, make small adjustments, and then observe what effect they have. That is the only way to develop an accurate and meaningful understanding of a complex system.

When viewing the systems you work in as "complex," in the Cynefin sense, the value of using observability becomes clear. Honeycomb is designed to enable that sort of probing and sensing. In the Q&A after his talk, "How Tracing Uncovers Half-truths in Slack's CI Infrastructure," Frank Chen described this process, quite delightfully, as “let's observe around and find out!” Similarly, in "Conditional Distributed Tracing" Will Sargent explained how software engineers often add additional spans to a Honeycomb trace to get a deeper sense of what's happening in a sensitive or complicated area of the system. In Cynefin terms, they are adding additional probes to make more sense of their complex code.

However, just probing and sensing is not enough if you want to truly grok a complex system—let alone make changes to that system to fix a bug, respond to an outage, or improve performance. You need to also take action based on what you’re seeing. In a complex system, you'll often be surprised to see that reality does not match your expectations of system behavior. Therefore, you must adjust what you're probing within the system, and eventually adjust the system itself. Several of our o11ycon+hnycon speakers devoted a good amount of their presentation time to describe their processes for doing that.
Iterating to understanding

Frank at Slack, Glen at CircleCI, and Michael at IMO all described kicking off their Honeycomb adoption by starting small, gaining some insights into the system, and then using those insights to make incremental enhancements. No one simply “added observability.” Achieving observability was an iterative process, with each step delivering insights that guide the next step in the journey.

Similarly, several speakers demonstrated that making changes to a complex system is best done incrementally—by following the process laid out in the Cynefin model to sense, probe, and respond. That process was a central theme of Michael’s "The Curious Case of the Latency Spike." The theme emerged as both Frank at Slack and Glen at CircleCI described pairing Honeycomb with progressive delivery techniques (such as feature flagging and canary launches) in order to make smaller, safer changes guided by feedback from their observability systems.

In fact, that concept of tight feedback loops comes up over and over again whenever engineers talk about how they practice observability. Simply observing a static system misses much of the value observability can provide. The true value of observability is in getting rapid, detailed feedback on the impact of changes to that system. The feedback cycle that allows a continuous cycle of small, incremental changes is sometimes referred to as a plan-do-check-act (PDCA) cycle, or an orient-observe-decide-act (OODA) loop. It’s a philosophy that the Lean community has used with great success to drive continuous improvement in complex systems.

It’s interesting to note that our o11ycon+hnycon speakers described two distinct flavors of this continuous improvement loop. First, it’s used to improve the systems they are operating. Second, the same approach of small, iterative changes was also held up as the best way to improve the observability system itself (how meta!).

For example, Frank at Slack described how feedback from observability tooling was used to drive a series of changes that pushed their testing “flake rate” from 50% to 5%. In addition, he described how the adoption of Honeycomb itself was also made incrementally. They explicitly did not start with a goal of “tracing all the things.” Instead, they took the approach of asking a question, figuring out what data was needed in Honeycomb to answer that question, and adding only what was necessary to get results.

Similarly, Michael described how IMO worked through their performance conundrums via an iterative cycle of decomposing their traces into more and more detailed spans, with each cycle of changes guided by the insights they’d gained from the previous cycle’s additions.
Read further for in-depth details

Seeing how engineers are tackling problems in the real world is always illuminating. It’s fascinating to see some common themes emerge from the very different contexts in which our various presenters operate. All our speakers described how observability enabled different parties to build a deeper understanding of their complex systems. There was a common narrative of starting small with curiosity, gaining insights, and then using those insights to drive the next change.

As you explore the details of the various Honeycomb use cases in this ebook, keep these themes in mind. They can provide useful insights as you think about how Honeycomb can help you understand your own complex systems.
The Curious Case of the Latency Spike

As presented by Michael Ericksen
Intelligent Medical Objects (IMO) is an electronic health record technology company that helps doctors better serve their patients. Time is often of the essence in healthcare settings, so when a spike in latency hit their application, IMO’s application team was on the case to solve the issues as quickly as possible.

But the breakthrough didn’t come from their most experienced engineers who’d been with the IMO for a while. It came from Michael Ericksen, a newly hired staff site reliability engineer, who explained the process of solving the mystery in his talk at the 2021 hnycon. By the time they found a solution, IMO had not only implemented observability, but also established a culture of what Michael described as “production excellence.”

The discovery: Actual latency in production didn’t match the models

When it came to latency, what IMO expected from its application didn’t match actual customer experience, and it wasn’t immediately clear why.

“IMO is primarily a serverless application that’s connected to an Elasticsearch back-end and is fronted by a load balancer,” said Michael. “Our application team has internal documentation with SLI-like (service-level indicators) statements about a target performance of 50 milliseconds per term for each item in the request payload. We are expecting request durations to scale more or less linearly with the number of items in the request payload.”

But when they looked at the actual application traffic, what IMO saw diverged from their models:

- At P50 (i.e., 50th percentile, or the mid-point of traffic load), 50% of requests took longer than 6.5 seconds to resolve.
- At P90 (90th percentile), 50% of requests were taking closer to 10 seconds to resolve.
- Almost all requests had between 20 to 30 items.

IMO expected to see the application take between 1 and 1.5 seconds to resolve requests containing 20 to 30 items, not 6.5 to 10 seconds. To get application performance within the parameters of the expected models, Michael and his team had to channel their inner Sherlock Holmes and start investigating.
The problem: The application was taking longer than expected to resolve requests.

The goal: Achieve the 1–1.5 second latency expected by the models, with 80% of requests completing faster than 50 milliseconds.

The red herring: Cold starts

One of IMO’s earliest theories was that the root cause of poor performance was the rate of cold starts. (A cold start is the initial start-up of the application. User requests after a cold start are considered “warm” and don’t take as long to get started.) IMO could see in their performance dashboard that the rate of cold starts was very high, almost matching all total function invocations. They posited that if they could reduce the rate of cold starts, they could improve request latency.

They ran an experiment where they paid to keep a certain number of instances warm, but this approach had only minimal impact on latency. The problem with this theory, IMO found, was that it costs money to keep instances warm in production and, for the money spent, it didn’t get the team close enough to their goal.

But in the process of trying to reduce the rate of cold starts, IMO’s application team came to an important realization.

“Whatever was happening, we couldn’t see it.”

IMO had an application performance monitoring (APM) tool that gave the team some visibility into application performance, but not the granular visibility they needed. They knew something was wrong, and, other than cold starts, the APM tool provided Michael’s team no other leads.

This situation is best illustrated in this graph Michael presented in his talk:
The orange area is uncaptured data in the telemetry of IMO’s APM vendor. IMO’s application is designed to handle network requests on background threads. Because auto-instrumentation from the APM vendor wasn’t capturing data on background threads, all of that performance (represented in orange) was therefore invisible.

To see what was happening in this orange area, IMO used Honeycomb to build out their observability capabilities to find more leads in the case.

“The way that we’re getting data into Honeycomb was by writing structured logs to CloudWatch and then ingesting them via a serverless function that’s deployed in our account. It’s an easy integration pattern as long as you’re able to write structured logs,” Michael explained. “You can ingest data into Honeycomb, and the documentation is really clear and easy to get up and off the ground. It also solved the primary issue that we experienced previously around multi-threading and the vendor agents.”
Following the leads: Refining the search using Honeycomb

But, according to Michael, implementing observability isn’t like a light switch; it’s a continual investment that has helped IMO narrow their search and eventually find the culprit. There were two big leads that Michael and his team were able to follow that led Michael to his final breakthrough.

Lead 1: First or second requests

For a single item, IMO made one request to Elasticsearch or two for rescoring logic. The IMO application team made an educated guess that the root cause of the latency issues could be within either the first or second request.

But breaking performance down within Honeycomb revealed that there wasn’t really any consistency in whether the first or second request was slower. In the graph below that Michael shared, the olive-green bars indicate the first request, while teal green indicates the second request.

Here is the breakdown of the latency for each request.

Whatever the problem was, it wasn’t unique to the first or second request and, as Michael put it, “Even our best and most experienced engineers guessed wrong.”
Lead 2: Elasticsearch itself

When digging into the second lead, Elasticsearch itself, the application team found that a round-trip search request took more than 13 seconds! That alone was worth noting, but the real issue was Elasticsearch reported that it took a single millisecond to process that 13-second search.

This situation was a perfect candidate for a custom attribute in a wide event because it helped eliminate Elasticsearch as the root cause. It wasn’t impacting the end user yet, but it might in the future so the team could start collecting data on it immediately to see how changes in the future might impact users.

But before they could dig much deeper into the second lead, Michael and another new team member had a breakthrough in the case.

The breakthrough: HTTP sessions and multi-threading

Knowing that Elasticsearch was reporting that searches were really fast, Michael and another engineer narrowed the focus to four lines of code.

```python
request = AWSRequest(
    method = 'Post',
    url = es_url,
    headers = headers,
    data = data).prepare()

http_session = URLib3Session()
raw = http_session.send(request)
respond = json.loads(raw._content)
```
Once their attention was focused on these lines, a few questions popped up:

- What if they moved the creation of the HTTP session outside of the multi-threaded context, which is where this code was originally executed?
- What if they moved it so that it executed once per function invocation or even once per function lifetime?
- What if they moved it out and shared it across the multi-threaded context?

So they moved it, ran some experiments with a shared HTTP session, and saw significant improvements. “Previously we were only seeing about 20% of requests completing faster than the target 50 milliseconds per term,” Michael said. “And with this simple change, we got this much closer to 40% of requests.”

They finally found the problem and, with some more experimentation, they were able to hit their goal of 80% of requests completing faster than 15 milliseconds. This gave them the confidence to deploy an update that reduced the number of threads that they were running on.

**Observability is about more than just solving mysteries**

As Michael mentioned, observability is not like flipping a switch. It’s an ongoing, step-by-step progression, and the best tools will help you ensure each step is thorough and may even identify your next step.

By properly implementing observability, engineers of all levels of experience can not only diagnose, but also solve major issues in production. Michael described this as more than just observability—it’s an attitude of production excellence.

“This observability thing [...] changed behavior for me. I started writing code differently. I started supporting code differently,” Michael explained during the Q&A portion of his talk. “It puts you on a trajectory—that some of the Honeycomb folks talk about—of production excellence. I think that’s a really valuable way to think about it. I don’t think the goal is necessarily observability. I think the goal is production excellence and delivering that value to your customers. Observability is a really important tool on that trajectory.”
The Unreasonable Effectiveness of a Single Wide Event

As presented by Glen Mailer
When running at scale, the typical small trickle of observability data can quickly become a flood. In those cases, sampling your trace data is recommended. But according to Glen Mailer, Senior Staff Software Engineer at CircleCI, not everything can or should be solved with tracing and sampling. As he explained in his talk at hnycon, the solution to production data volume issues can also be found using a single wide event as an alternative to how most tracing is currently done.

Initially, the benefits of using Glen's "single wide event" approach were presented as mostly budgetary—it's easy to wrap your head around how much an event costs (and therefore keep costs down) rather than trying to deal with dynamic call graphs of distributed tracing. But as his talk went on, a larger point became clear: using flexible tools like Honeycomb that reward this kind of out-of-the-box thinking allows you to experiment and find your own solutions.

### How a single wide event works

When you use a single wide event, you gather all the data related to a specific action into one event, which you then send to Honeycomb. Glen called them "single wide events" because they're not connected to a big, multi-part trace, which means they're not connected to a lot of other events. It's just one event with a lot of fields.

CircleCI bundles all data related to a business transaction into a single event, which they then send from their application to Honeycomb in one go. Glen said they keep 100% of these events.
If there are other services involved in servicing the transaction, CircleCI will send relevant transaction data from each service to the application, which then bundles it all up into one event to send to Honeycomb. Glen said they’ll either store that information in memory or in a database to pull later if it’s a longer transaction.

There are two major unique aspects to CircleCI’s single wide event process when compared to sampling with tools like Refinery:

1. Data from other services (arrows from A, B, C, etc) tend to be bespoke and are typically not standard tracing protocols.
2. Rather than sending a bundle of events over to Honeycomb, you’re sending one big one.

Why a single wide event is useful

Compared to dynamic tracing and sampling, budget for single wide events is easier to forecast and they can still answer a lot of production questions.

In an ideal world, you would store every single event for every single trace. For a system with significant traffic, this can be too expensive, so businesses typically turn to dynamic sampling.

But sampling and tracing can fail to reduce a budget if you’re not careful. As your system scales, you’ll need to add tracing and refine sampling capabilities alongside that growth. With wide events, as Glen put it, “[Two wide events are] not going to grow logarithmically as our system scales.”
In other words, it's just those two events. What grows as you scale are the fields you add to those events. Glen said that with this process, instead of managing tracing and sampling, you’re just adding more fields to an event and getting used to slicing and dicing it.

**CircleCI’s task-end event**

CircleCI sends a single wide event for each task they execute on the system called the “task-end event.” This event includes fields like:

- Build ID
- Build URL
- Customer
- Resource Class
- Executor
- Queue Time
- Run Time
- Pass/Fail
- Infrastructure Failure

CircleCI keeps 100% of task-end events, which are kept separate from dynamic sampling and don’t participate in the trace because CircleCI doesn’t set a trace ID. If they need to match later task-end events to the trace, they can do so with the Build ID and Build URL.

That approach aligns particularly well with their business needs. CircleCI’s core product is executing tasks—it’s what they charge people for. This direct relationship makes it easy to justify the cost of keeping 100% of the events is directly proportional to the revenue we’re charging to customers,” Glen explained. “This means we don’t have to worry about ‘Can we afford to keep all of these?’ We know we can, because there’s just one, and this is part of the core revenue stream.”

**Using task-end to fix infrastructure failures**

At CircleCI, an infrastructure failure is when a build goes down and it’s not the user’s fault. Glen described a situation where there was a spike in infrastructure failures—specifically, a spike of errors in the process of preparing an executor.

By looking at task-end, CircleCI was able to see into each transaction and identify that 95% of these errors originated from MacOS. Once the problem was identified, it was a simple matter of speaking with their MacOS data provider. An added benefit of task-end is that CircleCI was able to isolate each affected build and then manage the customer relationship to keep them happy.
Adding more single wide events

CircleCI adds single wide events to the most important actions in their system. There are two others Glen mentioned that fill in gaps task-end can’t address.

1. Task-start event: Each task-end event has an associated task-start event with similar fields. Together, they tell how long a build has been queued for.

2. Docker-pull event: This event is added because task-end only happens as a task finishes, which isn’t enough information to address issues around speed. If a transaction takes a while, it’ll be hard to understand what’s happened between task-start and task-end. So, CircleCI uses docker-pull to check in while a transaction is running and keep tabs on transaction speed.

A docker-pull event includes some of the fields task-end does, like Build ID and Build URL, and also includes unique fields like:

- Container Image
- Container Registry
- Pull Size, Duration, Speed
- Extract Size, Duration, Speed
- Create Time
- Errored?
- Resource Class
- Docker Version
- Etc.

Using docker-pull to fix speed issues

Using the docker-pull event, CircleCI was able to deal with an issue where clients in Asia were seeing very slow speeds. With docker-pull, they found out that many of these clients were using North American data centers, which was not a problem with CircleCI’s infrastructure. They used this information to update CircleCI’s documentation to warn users when they’re pulling from a region that’s inefficient.
Finding solutions doesn’t have to be expensive

These wide events are just one tool in CircleCI’s toolkit. Glen explained that they still use traces and this wide event solution is based on current Honeycomb pricing and processes. His ultimate point is that Honeycomb affords businesses a lot of flexibility in how they approach observability—a little out-of-the-box thinking is not only encouraged, but rewarded with a manageable budget and a unique tool for solving production issues.
How Tracing Uncovers Half-Truths in Slack’s CI Infrastructure

As presented by Frank Chen
Slack experienced meteoric growth between 2017 and 2020—but that level of growth came with growing pains. In his talk at the 2021 o11ycon+hnycon, Slack staff software engineer Frank Chen detailed one of Slack's biggest pain points in that period: flaky tests.

A flaky test returns both a passing and failing result despite no changes in the code. At one point, between 2017 and 2020, Slack's rate of flaky tests reached as high as 50%.

This amount of flakiness led to huge problems when it came to the DevOps practice of continuous integration (CI), where developers frequently integrate code into a central repository. As a result, developers' trust in testing was declining, developer velocity was starting to become sluggish, and huge incidents like a "large and cursed" Jenkins queue (as Frank described it) were starting to crop up.

So Frank set out to create a solution by applying observability through tracing to Slack's CI logic. Once fully implemented, Frank's approach helped reduce the rate of flaky tests per pull request (PR) down to 5%—a 10x reduction. Here's how he did it.

**Slack’s growth led to sluggishness and flakiness**

From 2017 to 2020, Slack evolved from a single web app with a monorepo to what Frank describes as "a topology of many languages, services, and clients that serve different needs." Most internal tools were built quickly and could scale just enough to keep up. This included the CI logic, which was built by the CTO and a handful of early Slack employees.

At around 7 minutes into his talk, Frank provided a rough overview of Slack's CI infrastructure from local branch development to testing. The Slack team found that, with a 10% month-over-month growth in test execution count, this CI infrastructure was starting to buckle.

As the rate of flaky tests started to reach 50%, Slack's internal tools teams started to hear two main complaints:

1. It is slow. Which, according to Frank, is the hardest problem to debug in distributed systems.
2. It is flaky. Which erodes developers’ trust in the efficacy of the systems.

To try and understand what was happening and why, the internal tools team at Slack got to work devising their own tracing solution, which Frank was later able to apply to the old CI logic.
Slack’s tracing solution: SpanEvents, SlackTrace, and Honeycomb

Suman Karumuri, Frank’s mentor and lead on the observability team at Slack, pioneered the development of Slack’s tracing solution, which includes a new data structure called a SpanEvent, an in-house solution called SlackTrace, and Honeycomb.

Suman did a full write-up here, which Frank recommended. Frank summarized Suman’s approach like this:

• First, Slack implemented a SpanEvent structure that allows them to create an event once and use it in multiple places.

• Then, Suman’s SlackTrace pipeline can ingest SpanEvents from multiple clients, allowing the Slack team to create views from the same data model by processing it through Kafka.

• Users can then access and analyze SpanEvents through a data warehouse or a real-time store, like Honeycomb.

Suman explains in his article that with this setup, Slack used their data warehouse with Presto to perform complex queries and Honeycomb to address issues in real time. In fact, their real-time store can provide access to trace data with a latency of fewer than 5 seconds. Honeycomb helps the team at Slack visualize and analyze this real-time trace data, which Suman says, “plays an important role in making our traces useful for triage.” Frank’s innovation is applying this setup and benefit to CI, thereby addressing the issues of both speed and flakiness.

How Frank applied tracing & observability to Slack’s CI

The CI infrastructure Frank inherited remained mostly unchanged for four years since it was established. By applying tracing to Slack’s CI infrastructure, Frank was able to transform it one Honeycomb-powered triage at a time.
Frank started small

Shortly after joining Slack, Frank had a conversation with Suman about SlackTrace. From that exchange, Frank was inspired to take the afternoon and create a quick prototype with their test runner.

During the PR rollout and a few simulated test runs, starting small allowed the team to rack up some quick wins. Frank noticed that a Git checkout step was performing slowly for a specific portion of their fleet. Using Honeycomb, they could see that a few instances within the auto-scaling group of underlying instances were not being updated. Knowing exactly where the problem was occurring, they were able to fix it quickly and easily.

Building on that momentum, Frank focused on gaining a full understanding of their entire CI process, explaining in the Q&A that, “one way I was able to understand how the pieces fit together was by using tracing to put in easy, small flags to understand: ‘Well, if we change how we initialize this part of our QA setup, does that affect anything else in that codebase?’”

As what John Casey at Red Hat might describe as “stakeholder zero” for Slack’s CI tracing, Frank had a responsibility to get buy-in from relevant stakeholders. Racking up quick wins is a great way to get momentum going and make iterative progress, but every once in a while a big, highly visible opportunity comes along.

Then he tackled the “Jenkins Queue, Large and Cursed”

A few months after his initial afternoon prototype, the turning point came for Frank on day two of a multi-day, multi-team incident. Frank described the situation as the “Jenkins Queue, Large and Cursed”:

“Day one, our teams were scrambling with one-off hacks to try to bring a few overloaded systems under control. On the morning of day two, I added our first cross-service trace by reusing the same instrumentation from our test runner.

Very quickly, with Honeycomb’s BubbleUp, it became clear where problems were coming from. On a portion of the fleet, we could see that Git LFS (Large File Storage) had slowed down the entire system. Over the next month, this sort of cross-system interaction led to targeted investments on how we can add this throughout Checkpoint traces.”
Specifically, Frank added traces to previously un-instrumented services at around 10 a.m. Results started streaming in through Honeycomb immediately. At around noon, Frank was able to diagnose the problem and the team could then get to work on fixing it. This quick diagnosis was only possible with Honeycomb visualizing that trace data, which one of Frank’s coworkers described as “dope as fuck.”

**Slack finally reduced flakiness with a unified effort**

One of the results of Frank’s CI tracing solution was a set of shared dimensions his team could use to make queries in Honeycomb legible and accessible. These dimensions were stubbed early in a library and instrumented with a few clients. Since then, various teams have extended and reused these dimensions for their use cases, creating a shared vocabulary for Slack’s CI tracing.

With this system in place, Frank’s team has been able to triage many CI issues quickly using Honeycomb. As they fixed each issue, iterating their code along the way, Slack was able to reduce test flakiness from 50% to 5%. This reduction has had a compounding effect on developer velocity and confidence in their production systems.

**Moving forward with CI tracing at Slack**

The keystone in Slack’s tracing system was Honeycomb, which allowed Frank to quickly triage problems as they occurred. Over time, the series of fixes introduced (thanks to Honeycomb) not only fostered excitement and buy-in for his efforts but also fixed long-standing problems with slowness and flakiness.

The importance of this solution is not lost on Frank, as early on in his talk he quoted Uncle Ben from Spiderman, saying “With great power comes great responsibility.” To learn more about how he wielded this “great power” [watch his full o11ycon+hnycon talk.](#)
Conditional Distributed Tracing

As presented by Will Sargent
Distributed tracing isn’t great. It’s good, but it has room to grow before it becomes truly great. Will Sargent, a software engineer at eero, encapsulated this reality perfectly in his talk at the 2021 o11ycon, where he described his proof of concept for conditional distributed tracing.

Distributed tracing is a method for monitoring application performance across a distributed system of microservices. As it is today, distributed tracing provides a number of solutions, which Will summarizes as:

1. Monitoring system health
2. Latency trend and outliers
3. Control flow graph
4. Asynchronous process visualization
5. Debugging microservices

But this last solution—debugging—is the most fraught with misunderstanding and competing priorities. Distributed tracing is a core component of observability. When a company adopts observability (and therefore tracing), many people need to use it—chiefly the software developers and the site reliability engineers (SREs)—in order for teams to fully realize the benefits.

Distributed tracing, up to this point in its history, has suited the debugging needs of SREs rather than developers. And forcing developers to adopt distributed tracing to debug can cause internal friction because they simply don’t need it.

The good news is that distributed tracing is relatively young. Enterprising developers such as Will are building on top of it using tools like Honeycomb to create new solutions. Will’s solution is conditional distributed tracing, which allows flexibility so that developers can change the way tracing works based on behavior code written by the application.
Debugging is a clear example of how tracing is still in its infancy and has significant growth potential. When it comes to debugging, developers are more like scientists who dig into hypotheses, testing out the logic behind the code. Site reliability engineers (SREs), on the other hand, are like firefighters trying to quickly locate the source of a fire, put it out, and apply what they learn to future fires. This difference in perspective leads to many of the issues around tracing adoption.

When debugging, developers look at the code. They put their effort into trying to understand the logic behind the issue that’s occurred. Will explains that their debugging process usually follows three general steps:

1. Create a pool of data from statements.
2. Fish around that pool with hooks and queries.
3. Keep the most useful statements for later.

The goal of this process is to resolve actual versus expected behavior. When something unexpected happens, developers want to understand why it happened. This process requires foresight on their part to predict the expected behavior. The way they compare their predictions versus what happened is usually with logs, which provide granular, but limited insight.

In this workflow, tracing spans aren’t very helpful. Will describes them as the new printkfs, where there is no priority system. The result is that using spans creates more data, more sampling, and more work for the developer. They know what they want to test, and spans provide more than is necessary.

Will summarized the prevailing attitude of developers to tracing: “Yeah, [tracing] is cool, but have you actually tried using it?” He pulled out a lot of examples in his talk, but one pull quote from Cindy Sridharan’s article, “Distributed Tracing—we’ve been doing it wrong,” stood out.

> Being able to quickly and cheaply test hypotheses and refine one’s mental model accordingly is the cornerstone of debugging. Any tool that aims to assist in the process of debugging needs to be an interactive tool that helps to either whittle down the search space or, in the case of a red herring, help the user backtrack and refocus on a different area of the system.

Cindy Sridharan, “Distributed Tracing—we’ve been doing it wrong”
Developers need a tool that helps them experiment with the code and its underpinning logic. Distributed tracing cannot accommodate this easily as it exists today because it can't easily prioritize data, and it often provides too much data for what's needed.

SREs, on the other hand, look at the whole system when they go about debugging. Their priority is to use debugging to locate the source of the problem that's occurred. The process SREs follow often centers around calling on multiple services to identify patterns and isolate the issue.

The important difference here is that, while developers are looking into why an issue is happening, SREs try to identify that an issue is happening. It’s the classic case of the unknown unknown, where something could be going wrong, and because you don’t know to look for it, you can’t fix it. You don’t know that you don’t know.

In this workflow, Will says that SREs “only use logs if they’re unsure about the mitigation strategy.” Logs only come into play when they’re not sure how to solve the issue. Instead, SREs rely on distributed tracing and sampling to understand the production environment to locate issues.

Both Charity Majors, CTO at Honeycomb, and Will agree towards the end of Will’s talk that locating the issue is 90% of the work to debugging. Despite the resistance of developers, adopting distributed tracing is essential if your business wants to effectively implement observability and reap its benefits.

So how do you strike a balance and adapt tracing to fit the needs of both your developers and SREs? Will’s answer is conditional distributed tracing.

**Conditional distributed tracing can address both perspectives**

Conditional distributed tracing is a proof of concept Will is working on that changes how tracing works based on behavior code written by the application. The application itself decides when and where it should produce a trace and when and where it should sample one. It’s not perfect, but it’s an iteration toward making distributed tracing more useful for both developers and SREs.
For developers, conditional distributed tracing brings them a step closer to being able to turn granular logging data on and off. The Microsoft Windows 11 team described the need for this capability in their paper “The Bones of the System: A Case Study of Logging and Telemetry at Microsoft.” With this ability, enabled via distributed tracing, developers can leverage tracing to test their hypotheses. They can try a solution, turn tracing on or off to see if it worked, and keep iterating.

For SREs, distributed tracing already does its job well, and conditional distributed tracing will help them get developers on board for their observability initiatives.

As he set about building his conditional distributed tracing proof of concept, Will defined three goals for himself:

- Allow behavior to depend on application-specific state.
- Augment spans and traces with additional information.
- Let developers take the wheel.

In achieving these goals, he hopes to make tracing more useful for everyone involved, developers and SREs alike. Let’s get into what he came up with.

## Conditional tracing proof of concept

Will admitted his proof of concept isn’t perfect, but it’s a good foundation that anyone can build upon.

Will’s solution uses the OpenTelemetry SDK hook driven by Groovy scripts. There are two systems: a conditional sampler and a conditional span builder.

- The conditional sampler allows for script-driven sampling, where you determine whether or not you want to sample a specific span.
- The conditional span builder allows for script-driven span creation, where the script determines whether you’re going to create a new span or run span.current.

Will goes on to explain that it’s not particularly fast, and he hasn’t done much work on making it secure, but the best option is typically to allow it to be a targeted feature flag so you have the ability to turn it on and off dynamically just as you might do for other features.
Once implemented, Will uses Honeycomb to keep an eye on how conditional distributed tracing is working. Overall, his solution for conditional tracing allows for:

- Flexibility on a different axis
- Application-based span control
- Exploration of OpenTelemetry SDK
- Answer the debugging problem for multiple audiences (SREs and developers)

Try Will’s solution for yourself at his Github: https://github.com/wsargent/conditional-tracing

You can also watch his session, where he walks through the proof of concept, how it works, where it can improve, and what’s next.

**Tracing is still in its infancy**

Conditional distributed tracing is just one example of the future of distributed tracing. We’re still in the infancy of this technology, and you can have a very real hand in shaping that future.

Build on Will’s work or create your own solution for another problem. Along the way, you can use Honeycomb to test, experiment on, and prove your ideas.
We sometimes talk about observability as a journey, but it's one that doesn't really have a final destination. As the stories in this ebook have illustrated, there's always more to discover about our systems and deeper insights to gain.

Learning from other engineers who are walking a similar path often provides inspiration for our own journey. We hope that you drew inspiration from the adventures of other engineering organizations and will be able to bring some ideas to bear in your own quest for observability. And if you're at the beginning of that journey, we encourage you to get started with Honeycomb for free today.
Honeycomb provides observability for modern development teams to learn, debug, and improve their production systems efficiently so that business-critical apps perform with minimal disruption to users. Honeycomb’s customers rely on the product for fast incident response, system optimization, and delivering pain-free releases that translate to happy devs and happy customers. Learn more at www.honeycomb.io and follow us on Twitter.

© 2021 Hound Technology, Inc.