GUIDE

# Achieving Observability

March 2021

honeycomb.io

# Table of Contents

# Introduction

Observability has been gaining traction recently as more and more people are finding out that their slew of monitoring tools and alerts are failing to surface problems quickly or accurately enough. Modern systems have complex distributed architectures, and legacy tools simply weren't designed to help users figure out the multitude of things that could possibly be wrong when failures occur. Today, the software industry needs to move beyond the troubleshooting guesswork created by the gap between modern architectures and yesterday's tools.

Observability is the solution to that gap.

This guide delves into observability—what it is (and isn't) and how an obscure mathematical term has been adapted to the practice of modern software development. We'll also cover how modern development teams can implement observability practices today so they can gain the confidence and understanding they need to create more performant and reliable production services.

# What is observability?

The term "observability" was first used in 1960 by engineer Rudolf E. Kálmán to describe mathematical control systems. From Wikipedia:

> *In control theory, observability is a measure of how well internal states of a system can be inferred from knowledge of its external outputs.*

When applied to software development, observability has been adapted to define a system where you have the ability to:

- Understand the inner workings of your application
- Understand any system state your application may have gotten itself into
- Understand the above, solely by observing its external outputs
- Understand that state, no matter how novel or bizarre
- Answer arbitrary questions about your environment without having to know ahead of time what you wanted to ask

## What makes a system observable

A system is "observable" to the extent that you can explain what is happening on the inside solely from what it tells you about itself from the outside. A key requirement is that you must be able to explain that *without having to add new instrumentation or write new code.* Adding new code or new instrumentation isn't wrong, but your system isn't *appropriately observable* until you don't have to add more to figure out anything you need to know about it.

## Appropriately observable?

A system that is *appropriately observable* is a system that is instrumented well enough that you can ask any question required to support it at the level of quality you want to deliver to your users. Observability isn't about mindlessly laboring over instrumenting every CPU instruction, every mouse move the user makes or how long it takes to add two numbers together. Instead, it's about understanding what level of service you need to deliver and instrumenting so you can meet those requirements.

We'll discuss some specific recommendations to instrument an appropriately observable system and its defining characteristics a bit later.

### What observability is not

Before we move on, there's another definition of observability we want to acknowledge and address: the definition where some conflate it as simply another synonym for monitoring or telemetry. Those who purport that misguided definition of "observability" will typically explain that it consists of three pillars: metrics, logs, and traces. That's simply the data, not its value.

Observability can't be achieved by gluing together disparate monitoring tools. Traditional metrics and monitoring are a fundamentally reactive approach that served the industry well in the past, when systems were much simpler.

The problem today is that many teams try to monitor their systems for every possible failure. Modern distributed systems fail in innumerable and unpredictable ways. You can't possibly monitor for every single failure mode. And you can't predict what those failure modes might even be because many of the ones you've seen in the past are likely to never occur again in the future.

The value observability provides goes well beyond just the basic data you need to get there.

## Why do we need observability?

It comes down to one primary fact: software architectures are becoming exponentially more complex. Developers and operators continue to adopt modern approaches to deploying software (e.g., coupling various SaaS components, relying on container orchestration platforms, using multiple platforms) due to their many benefits (scalability, maintainability, agility, etc.). But that approach to deploying software also brings with it several unintended consequences that make observability a necessary component for managing these systems.

Not too long ago, most system architectures were much simpler. You'd have a classic LAMP stack, maybe one big database, an app tier, a web layer, and a caching layer, with software load balancing.  You could predict where most of the failures might occur and craft a few expert dashboards that allowed you to deduce nearly every performance root cause analysis you might have over the course of a year. Your team didn't spend a lot of time chasing bizarre and novel problems arising from transient combinations of edge-case error conditions because finding out what was going on and why was achievable. It was possible to reason about where problems were occuring because you generally knew what things might go wrong and, if you were surprised, you could guess at the ones you hadn't predicted in advance.

But now, with multiple platforms, the ability to operate and use infrastructure across various geographic locations with the click of a button, or a microservices architecture, it's commonplace to support applications with millions of unique users completing transactions that are sprawled across dozens of apps. When trying to figure out where problems are occuring, you may have a long, fat tail of unique questions to answer **all the time**. There are many more potential combinations of things going wrong, and sometimes they sympathetically reinforce each other. Before, you knew where the surprises might be lurking. Today, there's no way to know until it's a problem; in other words, there are many more **unknown-unknowns**.

When environments are as complex as they are today, simply monitoring for known problems doesn't address the growing number of new issues that arise. Many new issues are likely transient—you didn't know that a particular combination of failures was even possible and, after this problem is solved, you'll never likely see that exact problem again. In the modern world, unknown-unknowns are a more common issue, meaning that without an observable system, you don't know what is causing the problem and you don't have a standard starting point / graph to find out.

This escalating complexity is why observability is now necessary to build and run today's infrastructure and services.

## Where does monitoring fit in?

According to the *Oxford Pocket Dictionary of Current English*, monitoring is:

> *To observe and check the progress or quality of (something) over a period of time; keep under systematic review.*

That makes sense in the context of operations—you are checking the status and behaviors of your systems against a known baseline to determine if anything is not behaving as expected. Monitoring systems collect, aggregate, and analyze periodic metrics to systematically sift through known patterns that indicate failures might be occurring. Observability takes a different approach that allows you to identify new and unexpected failures.

Monitoring relies heavily on predicting how a system may fail and then checking to see if it has failed in that exact way. Monitoring systems then often generate large grids of dashboards that sit on your desktop or your wall, and give you a quick sense of overall system health at a glance. You can then check—or create and then check—any number of metrics to see if your system is performing within known good thresholds. Dashboard and metrics (i.e., monitoring) tools are terrific for understanding the state of things you know might go wrong—or the known-unknowns—in your system.

But monitoring doesn't help you:

- When you're experiencing a new and unexpected serious problem, and you're in the dark for hours until your customers tell you there's a problem
- When customers are complaining, but all of your dashboards are all green
- When something new fails and you don't know where to start looking

Monitoring doesn't help you solve these problems because they are fundamentally reactive and the views they provide are too coarse to see granular levels of individual detail.

The basis of monitoring systems are metrics. A metric is a numerical representation of system state over the particular interval of time when it was recorded. Similar to looking at a physical gauge, at a glance, a metric might be able to convey whether a particular resource is over- or under-utilized at a particular moment in time. For example, CPU utilization might be at 90% right now or p95 latency is below 200ms.

Metrics are pre-aggregated data. That means that by the time you see that measure, all of the underlying system state responsible for generating it has been aggregated into that one number that is reported. While those pre-aggregated, or write-time, metrics are efficient to store and fast to query, using them means you lose any of the underlying context to understand **why** that number was reported. You can't understand why, for example, CPU utilization was at 90% unless you knew in advance that you should also be capturing metrics for the number of concurrent process threads, incoming requests, buffer pool sizes, or any of the other hundreds of reasons that could be happening.

As a result, what often happens is that teams get into an ever-escalating arms race of collecting thousands upon thousands of metrics in an unending game of whack-a-mole. When you finally whack one problem and start monitoring for its associated metrics, another one surprisingly pops up, and the cycle starts all over again.

This fundamentally reactive approach worked well enough for traditional monolithic systems all running on the same server rack because there was a relatively finite number of failure combinations that could happen—a problem was either on the app servers or in the database. And you often knew there was a problem because a service was completely down and everyone experienced the same problem, and aggregate measures reflected that.

In today's world of modern distributed and resilient systems, slow is the new down. Total failures are less common, yet small hidden problems constantly happen. When your customers have a problem, they don't care about overall system health or your p99 latency. They care about what **they** are experiencing.

But monitoring systems can't show you that. Still, many teams try to make monitoring systems fit that purpose. Before they know it, they're spending more time setting up new and increasingly elaborate dashboards to see a new type of problem that will likely never reappear. Now an impenetrable thicket of similarly named dashboards exists and teams must wade their way through them all because pruning them becomes impractical—what if that one error we saw that one time comes back again?

Metrics systems also have a difficult time handling highly unique (or high-cardinality) data like user, request ID, shopping cart ID, source IP, etc. Capturing metrics for fields with millions of unique fields (each with their own subset of unique dimensions) creates an exponential explosion of metrics data. Many systems simply can't support that type of data load or, if they can, it's impractically slow to query and prohibitively expensive to store.

Monitoring can, however, provide a great view of infrastructure level system state; things like memory, disk, and CPU. It can reveal physical operating constraints that are best to understand in aggregate—for instance, is your overall request bandwidth close to maxing out the capacity of your network cards? But to understand application-level performance, where it's necessary to drill down to see how individual customers or transactions are experiencing your services, observability can help in a more proactive way.

## Is it time to automate everything? What about AI and Machine Learning?

That escalating game of metrics whack-a-mole leads some to conclude that perhaps the next right answer is to start down the path of the Gartner-termed "AIOps" approach. The promise of AIOps is that AI and Machine Learning (ML) can be used to drive better decision-making and faster incident resolution by mitigating many of the "people problems" involved in managing complex systems.

In the context of any consideration of AI / ML, it's important to note that the technology isn't magic. AI accomplishes tasks similar to tasks humans accomplish, just much faster and more patiently. We recommend reading Danyel Fisher's primer on the misleading promise of AIOps for an in-depth look at the challenges AIOps needs to overcome.

A fundamental principle for any AI system is that it needs to be trained with example data that helps it find a boundary between “good” and “bad” examples on which to make decisions. But, again, this approach is fundamentally reactive—it relies on identifying known-unknowns, and it is poorly equipped to detect new problems. A popular workaround to address that is to instead have AI systems focus on anomaly detection.

The challenge is that all sorts of intentional anomalies occur and that not all anomalies can be detected. Consider the some of these scenarios:

- Deployments are anomalies. When continuous deployments are happening, every new feature or bug fix shifts the previously known pattern of performance and behavior. Is today’s sudden drop in resource usage at noon, compared to yesterday’s performance at noon, actually a problem? Your AI system will think so and trigger an alert. In a world where dozens of trivial and constant adjustments are made, you’ll still be generating noisy and unactionable alerts.
- Canadian users of your app running a French language pack on a particular version of iPhone hardware encounter a firmware condition that prevents your app from saving files to local cache. Everything appears to serve without error, but to your customers, it FEELS like photos are loading very slowly. Your AI system detects no errors and does not trigger any alerts.
- A new feature deployment causes additional sequential database queries when users have enabled an optional feature. This is working as intended, but not as desired. How would your AI system even detect that?

## Strengthen the human element

In the above scenarios, AI and ML systems could potentially identify some of those outlier conditions, but they cannot reasonably decide if those outliers are actually a problem. Instead of relying solely on human pattern-detection power or solely on machine decision-making power, what’s needed is an approach that allows humans and machines to both do what each does best.

Machines are able to quickly sift through billions of rows of data to identify outliers and related anomalies, but they lack the ability to make meaningful decisions when lacking context about the intent of humans. Humans excel at quickly making correct contextual decisions about desired outcomes, but lack the ability to quickly sort through mounds of data. Rather than automating everything away in the misleading world of AIOps, observability instead applies analytical machine capabilities in ways that strengthen the human element.

Observability is designed to operate something like a mecha-suit—a robot-powered exoskeleton with a human at the controls. Observability gives people machine-based superpowers that extend their troubleshooting senses, support their intuitions, make it quick and easy to disprove or explore hypotheses, and move at unparalleled speed to find the real source of issues.

# Who is observability for?

Observability is for anyone who cares about the way that code operates in production to serve their customers. Increasingly, the titles of the people to whom this applies aren't always clearly defined or consistent from organization to organization. DevOps has popularized the idea that developers should care more about operational concerns and that operators should care more about development concerns.

The reality is that observability can help anyone that needs visibility into what's happening in production. But let's start with the first two roles we often hear about: developers and operators. Who is observability for?

## Observability is for developers *and* operators

Observability approaches troubleshooting by answering questions about your system using data, an ability that is as valuable for developers as it is for operators. This is especially important to highlight since one of the responses to the DevOps cultural movement has been to counter that collaborative approach with movements like "No-Ops" that advocate for just removing operational considerations entirely.

In "The Future of Ops Careers," Honeycomb CTO and Co-founder Charity Majors takes an in-depth look at the changing nature of operations roles. Even in serverless environments, if you ship software, you have operations work to be done that ensures production services are running smoothly. As more companies adopt virtualized and abstracted platforms to run their software, the traditional operational duties like monitoring server clusters instead become modern operational practices like smoothing out platform integrations and focusing on service reliability. These teams need proactive, fine-grained solutions to detecting and resolving complex and novel problems in production systems.

At the same time, developers are also now increasingly tasked with owning the operation of their code in production, but often, nothing has been done to help them see how the practices they use to understand code operability apply to the world of managing services in production.

In "The Future of Developer Careers," Honeycomb CEO and Co-founder Christine Yen closely examines how developers often are given tools like metrics and dashboards but without context that connects that information back to the code, business logic, or customer needs that are the world of software development.

Observability is what bridges that gap for both developers and operators. Operators are using observability to help uplevel developers who are new to production by reducing the stress of being on-call by arming them with the tools they need to quickly resolve production incidents. Developers are using code-oriented observability tools to teach operators how to dig into their code, methodically walk through logic, and use instrumentation to question assumptions and validate hypotheses.

# What you need to leverage observability

## Speed

When investigating a problem in production, you need fast performance:

- To send data quickly so it can be queried. If your ETL takes 10 minutes, it's too long.
- To query large amounts of **high-cardinality** data quickly. When investigating an issue with a production service, you need to be able to get results back in sub-seconds, not minutes.
- To rapidly iterate through hypothesis after hypothesis to explore the various potential sources of the problem.

The **cardinality** of a given data set is the relative number of unique values in a dimension. For example, if you have 10 million users, your highest possible cardinality field / dimension is probably the user ID. In the same data set, user last names will have lower cardinality than unique ID. Age will be a low-cardinality dimension, while species will have the lowest cardinality of all: {species = human}.

Exploring your systems and looking for common characteristics **requires support for high-cardinality fields as a first-order, group-by entity**. When you think about useful fields you might want to break down or group by, all of the most useful fields are usually high-cardinality fields because they do the best job of uniquely identifying your requests. Some to consider are UUID, app name, group name, shopping cart ID, unique request ID, and build ID. All are incredibly useful for troubleshooting and all are high cardinality.

## Breadth-first and depth-first investigation options

An **event** represents a unit of work in your environment.

An event can tell a story about a complex thing that happened—for example, how long a given request took or what exact path it took through your systems. When your application emits events, it should emit them with as much context as possible so the person working on investigating can be better informed. At minimum, an event should contain information about the process and host that emitted it, and the time at which it was emitted. Additionally, record request, session, and user IDs should also be provided, if applicable and available. More context is always better than less, and filtering context out is a lot easier than injecting it back in later.

A series of related events can be stitched together as a **trace**. Tracing shows the relationships among various services and pieces in a distributed system, and tying them together helps give a more holistic view of what's happening in production. Another way of thinking about it is that traces are really just a series of related events.

To save on resources, it can be tempting to aggregate, average, or otherwise roll up events into metrics before sending them to whichever system you use to understand the state of your production systems. But if you do that, you lose the ability to delve into finer-grain details than the aggregate view you recorded. All that now exists in that rolled-up view is one coarse measure of behavior over a period of time. Aggregated views of behavior prevent your ability to answer the fine-grained questions that can help you solve a problem.

With wide, context-rich events, you have a broad range of dimensions to start slicing by to get to the solution. Is the service failing requests on all endpoints or just a handful? Are all regions affected? Does it happen for everybody or just Android users? And once you zoom in, you have individual events that stand alone. Events provide a granularity that lets you notice how individual customers, or subsets of customers, experience your services. (We'll talk more about what goes into an event and how to think about building them later.)

Humans need wide events and deep traces to:

- Have access to as much rich context as possible when troubleshooting a problem.
- Answer questions without having to re-instrument the service or write new code.
- Be able to switch back and forth among different visualizations and levels of granularity without having to switch interfaces (and lose context).

### Collaboration

You can achieve observability without collaboration, but if you take this route, you won't be able to effectively leverage what you learn or keep up with the accelerating rate of change and complexity. We recommend choosing tools that enhance your team's ability to:

- Leverage what the entire team knows about a given problem or service.
- Elevate the entire team to the level of the best debugger / problem solver.
- Onboard new team members quickly and effectively with real data.
- Save time by not having to reinvent the wheel every time a new investigation begins.

## Observability-Driven Development practices

That escalating game of metrics whack-a-mole leads some to conclude that perhaps the next right answer is to start down the path of the Gartner-termed “AIOps” approach. The promise of AIOps is that AI and Machine Learning (ML) can be used to drive better decision-making and faster incident resolution by mitigating many of the “people problems” involved in managing complex systems.

In the context of any consideration of AI / ML, it's important to note that the technology isn't magic. AI accomplishes tasks similar to tasks humans accomplish, just much faster and more patiently. We recommend reading Danyel Fisher's primer on the misleading promise of AIOps for an in-depth look at the challenges AIOps needs to overcome.

## A note on Observability-Driven Development (ODD)

Observability isn't just for operations folks or just for "production incident response." It's also for answering the day-to-day questions we have about our systems, so the people who build and maintain the systems can form hypotheses, validate hunches, and make informed decisions all the time and not just when an exception is thrown or a customer complains.

In simple terms, ODD is somewhat cyclical, looping through these steps:

1. What do your users care about? Instrument to measure that thing.
2. Now that you are collecting data on how well you do the things your users care about, use the data to get better at doing those things.
3. Bonus: Improve instrumentation to also collect data about user behaviors—such as how users try to use your service—so you can inform feature development decisions and priorities.

Once you have answered at least the first question in this list, you are ready to make strides toward achieving observability.

During the development process, lots of questions arise about systems that are not "production monitoring" questions, and aren't necessarily problems or anomalies—they're about hypotheticals, specific customer segments, or "what does 'normal' even mean for this system?" Developers need observability to answer those questions.

Configure your observability tools to use the nouns that are already ingrained into software development processes—build IDs, feature flags, customer IDs—so developers can move from *"Oh, CPU utilization is up? I guess I'll go … read through all of our benchmarks..."* to *"Oh! Build 4921 caused increased latency for that high-priority customer you've been watching? I'd better go take a look and see what makes their workload special."*

# Achieving observability

To achieve observability, the first step is, well, simply taking that first step. Don't think too far ahead and fall prey to analysis paralysis. Don't think about how far you have to go or how much work there is to do. Just start, and tomorrow, you will have greater observability than you do today.

STEP ONE

## Define what you need to achieve

Before engaging in a major instrumentation project, determine what "appropriately observable" means for your business. For instance, what level of observability do you need to deliver the quality of service that's expected? What do your users care about? What will they notice?

> As discussed earlier, "A system that is appropriately observable is a system that is instrumented well enough that you can ask any question required to support it at the level of quality you want to deliver to your users."

STEP TWO

## Understand your priorities

Overall, with observability, the health of each end-to-end request is of primary importance versus the overall health of the system. Context is critically important because it provides you with more ways to see what else might be affected or what the things going wrong have in common. Ordering is also important. Services will diverge in their opinion of where execution time was spent.

The health of each high-cardinality slice is of next-order importance—how is the service performing for each user, each shopping cart, each region, each instance ID, each firmware version, each device ID, and any of them combined with any of the others.

Infrastructure health matters in that it impacts software performance. How much you need to monitor infrastructure health depends on how much visibility and control you have over your code's underlying hardware (is it on-prem, in the cloud, serverless, etc?). For monitoring infrastructure level health, metrics (i.e., monitoring) do a good job of informing you about problems that affect software performance. The "Getting Started with Honeycomb Metrics" whitepaper provides an in-depth exploration of how metrics and observability come together.

STEP THREE

## Make good events from the code / logs you don't write yourself

When you can't instrument the code yourself (e.g., the cloud services you use), look for the instrumentation provided and find ways of extracting the information you need. For example, there's only so much you can tell about database internals without modifying its source code. If you wanted to achieve observability for your MySQL database, you would need a couple of different approaches. The overall volume of MySQL events are generally identical, so they would need to be sampled (see the Sampling section later in this guide). You would also want to tail the slow query log to alert you of potentially problematic queries. It would also be useful to periodically run MySQL commands to capture InnoDB performance stats and queue lengths. Combined, these approaches would give you a granular, event-based view of what was happening inside your database.

### Build events that tell a story

An event is a record of something that your system did. A line in a log file is typically thought of as an event, but events can (and typically should) be a lot more than that—they can include data from different sources, fields calculated from values from within or external to the event itself, and more.

Many logs are only portions of events, regardless of whether those logs are structured. It's not at all uncommon to see 5-30 logs that together represent what could usefully be considered one unit of work. For example, logs representing a single HTTP transaction often go something like:

```
6:01:00 accepted connection on port 80 from 10.0.0.3:63349
6:01:03 basic authentication accepted for user foo
6:01:15 processing request for /super/slow/server
6:01:18 request succeeded, sent response code 200
6:01:19 closed connection to 10.0.0.3:63349
```

The information contained in that block of lines—i.e., that collection of log messages—is all related to a single event: the handling of that one connection. But instead of being helpfully grouped into one event, it's spread across many disconnected log messages. Sometimes there's a request ID that lets you connect them back together. But there isn't most of the time, and you have to sort through PIDs and remote ports, and sometimes there's just no way to put all those pieces back together. So if you can avoid that, you should. If you're in this situation, you should investigate the work required to invest in logging better events.

### The changing scope of an event

An event should have everything about what it took to perform that unit of work. This means it should record:

- The input necessary to perform the work.
- Attributes computed, resolved, or discovered along the way.
- The conditions of the service as it was performing the work.
- Some details on the result of that work.

Treating an event as a unit of work lets you adjust its scope, depending on the goals of the observer. Sometimes a unit of work is downloading a single file, parsing it, and extracting specific pieces of information. Other times, it's getting an answer out of dozens of files. Sometimes a unit of work is accepting an HTTP request and doing everything necessary to render a response. Other times, one HTTP request can generate many individual units of work that are more useful to consider as discrete events.

The scope of what is considered one unit of work can change as needed to observe different parts of a service, and the instrumentation you have in that service can change to accommodate those needs. Altering that scope allows you to zoom in on troubled areas and back out again to understand the overall behavior of the entire service.

STEP FOUR

## Instrument the code you write yourself

Start with a service written in a language that has pre-built support for automatic (i.e., out-of-the-box) instrumentation. Auto-instrumentation offers you a way to easily start by providing basic timings and traces so you can instead spend your time instrumenting the functionality you care most about within your own custom applications. The next section describes some principles and specific recommendations for evaluating and instrumenting your code.

# Instrumenting for observability

What telemetry data is actually useful to capture via instrumentation is, of course, dependent on the details of your service. That said, most services can get something out of these suggestions.

## Broad instrumentation principles

**Generate unique request IDs** at the edge of your infrastructure, and propagate them through the entire request lifecycle, including to your databases (in the comments field). See our Tracing Guide for more details on why this matters.

**Generate one event per service / hop / query / etc**. For example, a single API request should generate a log line or event at the edge (ELB / ALB), the load balancer (Nginx), the API service, each microservice it gets passed off to, and for each query it generates on each storage layer. There are other sources of information and events that may be relevant when debugging—for example, your database likely generates a bunch of events that say how long the queue length is and reporting internal statistics or you may have a bunch of system stats information—but one event per hop is the current easiest and best practice.

**Wrap any call out to any other service / data store as a timing event**. Finding where the system has become slow can involve either distributed tracing or comparing the view from multiple directions. For example, a DB may report that a query took 100ms, but the service may argue that it actually took 10 seconds. They can both be right—if the database doesn't start counting time until it begins executing the query and it has a large queue.

**Collect lots of context**. Each event should be as wide as possible, with as many high-cardinality dimensions as possible, because this gives you as many ways to identify or drill down and group the events and other similar events as possible.

**Add redundant information** when there's an enforced unique identifier and a separate field that is easier for the people reading the graph to understand. For example, perhaps in a given service, a Team ID is globally unique and every Team has a name.

**Add two fields for errors**. The error category and the returned error itself, especially when getting back an error from a dependency. For example, the category might include what you're trying to do in your code (error reading file) and the second what you get back from the dependency (permission denied).

**Opt for wider events (more fields)** when you can. It's easier to add in more context now than it is to discover missing context later. When in doubt, just add in more context. With systems like Honeycomb, there's no penalty for making arbitrarily wide events (one event with 10 fields or 1000 fields is still just one event).

**Don't be afraid to add fields that only exist in certain contexts**. For example, add user information if there is an authenticated user, don't if there isn't. You don't need to have the same fields in every event.

**Be thoughtful about field names**. Is that name descriptive and unique? When troubleshooting, you'll want as much easily understandable context provided as possible. Common field name prefixes help when skimming the field list if they're alphabetized.

**Add units to field names**. What unit is the value in? Is that 8 seconds or milliseconds? Is that 100MB or 100GB? Use fields that explicitly define the units your measures capture, such as parsing_duration_µs or file_size_gb.

## Specific field recommendations

Adding the following to your events will give you additional useful context and ways to break down the data when debugging an issue.

### Who's talking to your service?

- Remote IP address (and intermediate load balancer / proxy addresses)
- If they're authenticated
  - user ID and user name (or other human-readable identifier)
  - company / team / group / email address / extra information that helps categorize and identify the user
- user_agent
- Any additional categorization you have on the source (SDK version, mobile platform, etc.)

### What are they asking of your service?

- URL they request
- Handler that serves that request
- Other relevant HTTP headers
- Did you accept the request? Or was there a reason to refuse?

- Was the question well formed? Or did they pass garbage as part of the request?
- Other attributes of the request. Was it batched? Gzipped? If editing an object, what's that object's ID?

### How did your service deal with the request?

- How much time did it take?
- What other services did your service call out to as part of handling the request?
- Did they hand back any metadata (like shard, or partition, or timers) that would be good to add?
- How long did those calls take?
- Was the request handled successfully?
- Other timers, such as around complicated parsing
- Other attributes of the response—if an object was created, what was its ID?

### Business-relevant fields

Business logic fields are optional, as this type of information is often unavailable to each server. But when available, it can be super useful in terms of empowering different groups to also use the data you're generating. Some examples:

- Subscription tier: Is this request related to an account in the bronze, silver, or gold-level plans you provide?
- Specific service-level agreements (SLAs): If you have different SLAs for different customers, including that information inside a request can let you issue queries that take it into account.
- Account rep name, business unit, etc.

These types of fields aren't necessarily useful in tracking down production issues, but they can help with business-intelligence use cases when the performance of specific users or groups of users needs to be clearly understood.

### Additional context about your service / process / environment

- Hostname or container ID or …
- Build ID
- Environment, role, and additional environment variables
- Attributes of your proceses—e.g., amount of memory currently in use, number of threads, age of the process, etc.
- Your broader cluster context (AWS availability zone, instance type, Kubernetes pod name, etc.)

# Using sampling to manage the volume of data

Before we get started in this section, it's important to note that you **do not need** to sample your data to work with observability tools. For the vast majority of applications, the incremental cost of collecting observability data is incredibly minimal and unobtrusive.

But when you're trying to achieve observability at scale (think: *hundreds of billions of events per year*), suddenly every byte generated can add up to immense amounts of data and traffic to manage. When your stream of observability data instead turns into a raging flood, there's a tradeoff to consider: Exactly how much data do you need to achieve the desired result, without breaking other systems or the bank? At that scale, you may **want to** sample your data.

At that type of scale, we recommend sampling your data to control costs, prevent system degradation, and to encourage thinking of data in terms of what is important and necessary to collect.

Sampling is the idea that you can select a few representative elements from a large collection and learn about the entire collection by looking at them closely. Beyond observability, sampling is a widely used tactic whenever trying to tackle a problem of scale. For example, a survey assumes that by asking a small group of people a set of questions, you can learn something about the opinions of the entire populace.

As a basic technique for instrumentation, sampling is no different—by recording information about a representative subset of requests flowing through a system, you can learn about its overall performance. And as with surveys, the way you choose your representative set (the sample set) can greatly influence the accuracy of your results.

Beyond problems of scale, sampling can sometimes also benefit your approach to observability by encouraging you to consider your telemetry holistically. Sampling techniques are often first explored as a means to control resource costs (otherwise, why potentially risk losing data fidelity?). However, all operational data should be treated as though it's sampled and best-effort, as opposed to merely being constrained by cost-benefit analysis. At any scale, sampling your data will create muscle memory around considering which parts of your data are actually important, not just important-ish. Like reducing alert noise by pruning unhelpful alerts, curating sample rates is a more advanced technique that can highlight the aspects of your system that truly matter the most to your business.

There are two main ways to approach sampling: constant or dynamic. Dynamic sampling offers greater flexibility.

## Constant sampling

Constant sampling means you submit one event for every *n* events you wish to represent. For example, if you're submitting 25% of all your events, you have a constant sample rate of 4. Your underlying analytics system can then process this kind of sampling very easily: multiply all counts or sums by 4 to reconstruct a representative view of all traffic. (Averages and percentiles are weighted accordingly.)

**The advantage of constant sampling** is that it is simple and easy to implement. You can easily reduce the load on your analytics system by only sending one event to represent many, whether that be one in every four, one hundred, or ten thousand events.

**The disadvantage of constant sampling** is its lack of flexibility. Once you've chosen your sample rate, it is fixed. If your traffic patterns change or your load fluctuates, the sample rate may be too high for some parts of your system (missing out on important, low-frequency events) and too low for others (sending lots of homogenous, extraneous data).

**Constant sampling is the best choice when your traffic patterns are homogeneous and steady**. If every event provides equal insight into your system, then any event is as good as any other to use as a representative event. The simplicity allows you to easily cut your volume.

## Dynamic sampling

Dynamic sampling means you vary the sample rate based on characteristics of the incoming traffic. This sampling approach allows you tremendous flexibility in how you choose the individual events that will be representative of the entire stream of traffic.

### Dynamic sampling: A static map of sample rates

Building a static map of traffic type to sample rate is the one method for doing dynamic sampling. It allows you to enumerate different types of traffic and encode different sample rates for each type in your code. The varied sample rates represent your differing values for different types of traffic.

By choosing the sample rate based on an aspect of the data that you care about, you gain more flexibility to control both the total volume of data sent and its resolution.

**The advantage of a static map of sample rates** is that you gain flexibility in determining which types of events are more important for you to examine later while retaining an accurate view into the overall operation of the system. If you know that errors are:

- More important than successes, or
- Newly placed orders are more important than checking on order status, or
- Slow queries are more important than fast queries, or
- Paying customers are more important than those on the free tier,

then this is a method that helps manage the volume of data you send into your analytics system while still gaining detailed insight into the portions of the traffic you really care about.

**The disadvantage of a static map of sample rates** is that if there are too many different types of traffic, enumerating them all to set a specific sample rate for each can be difficult. Additionally, you must enumerate the details of what makes traffic interesting when creating the map of traffic type to sample rate. Though this method provides more flexibility over a constant rate, it can be difficult to create the map if you don't know ahead of time which traffic types might be important. If traffic types change their importance over time, this method can't easily change to accommodate that.

**This method is the best choice when your traffic has a few well-known characteristics that define a limited set of types**. Some types are obviously more interesting for debugging than others, and some common patterns for using a static map of sample rates are HTTP status codes, error status, top-tier customer status, and known traffic volume.

## Key-based dynamic sampling

The key you use to determine the sample rate can be as simple (e.g., HTTP status code or customer ID) or complicated (e.g., concatenating the HTTP method, status code, and user-agent) as needed to select samples that can give you the most useful view into the traffic possible.

The following methods all work by looking at historical traffic for a key and using that historical pattern to calculate the sample rate for that key.

## Constant throughput

For the constant throughput method, you specify the maximum number of events per time period you want to send. The algorithm then looks at all the keys detected over the snapshot and gives each key an equal portion of the throughput limit. It sets a minimum sample rate of 1 so that no key is completely ignored.

**Advantages to the constant throughput method**. If you know you have a relatively even split of traffic among your keys and that you have fewer keys than your desired throughput rate, this method does a great job of capping the amount of resources you will spend sending data to your analytics.

**Disadvantages to the constant throughput method**. This approach doesn't scale at all. As your traffic increases, the number of events you're sending in to your analytics doesn't. As a result, your view into the system gets more and more coarse, to the point where it will barely be useful. If you have keys with very little traffic, you risk under-sending the allotted samples for those keys and wasting some of your throughput limit. If your keyspace is very wide, you'll end up sending more than the allotted throughput due to the minimum sample rate for each key.

**This method is useful as a slight improvement over the static map method** because you don't need to enumerate the sample rate for each key. It lets you contain your costs by sacrificing resolution into your data. However, it breaks down as traffic scales in volume or in the size of the key space.

### Constant throughput per key

This approach allows the previous method to scale a bit more smoothly as the size of the key space increases (though not as volume increases). Instead of defining a limit on the total number of events to be sent, this algorithm's goal is a maximum number of events sent per key. If there are more events than the desired number, the sample rate will be set to correctly collapse the actual traffic into the fixed volume.

**Advantages to the constant throughput per key method**. Because the sample rate is fixed per key, you retain detail per-key as the key space grows. When it's simply important to get a minimum number of samples for every key, this is a good method to ensure that requirement.

**Disadvantages to the constant throughput per key method**. In order to avoid blowing out your metrics as your keyspace grows, you may need to set the per-key limit relatively low, which gives you very poor resolution into the high-volume keys. Additionally, as traffic grows within an individual key, you lose visibility into the details for that key.

**This method is useful for situations where more copies of the same error don't give you additional information** (but the error is still happening). You want to make sure that you catch each different type of error. When the presence of each key is the most important aspect, this method works well.

### Average sample rate

This approach tries to achieve a given overall sample rate across all traffic and capture more of the infrequent traffic to retain high-fidelity visibility. By increasing the sample rate on high-volume traffic and decreasing it on low-volume traffic, the overall sample rate remains constant and means you can catch rare events and still get a good picture of frequent events.

The sample rate is calculated for each key by counting the total number of events that came in and dividing by the sample rate to get the total number of events to send. Give each key an equal portion of the total number of events to send and work backwards to determine what the sample rate should be.

**Advantages to the average sample rate method**. When rare events are more interesting than common events and the volume of incoming events across the key spectrum is wildly different, the average sample rate method is an excellent choice. Picking just one number (the target sample rate) is as easy as constant sampling, but you also get good resolution into the long tail of your traffic while still keeping your overall traffic volume manageable.

**Disadvantages to the average sample rate method**. High-volume traffic is sampled very aggressively, which may not be desirable.

**This method is useful for situations where high volume traffic is largely successful**. When you're not concerned about sampling rates affecting very large volumes of traffic, this can be a useful strategy.

### Average sample rate with minimum per key

This approach combines two previous methods:

1. Choose the sample rate for each key dynamically
2. Choose which method you use to determine that sample rate dynamically.

One disadvantage of the average sample rate method is that if you set a high target sample rate but have very little traffic, you will wind up over-sampling traffic you could actually send with a lower sample rate. If your traffic patterns are such that one method doesn't always fit, use two. For example:

- When your traffic is below 1,000 events per 30 seconds, don't sample.
- When you exceed 1,000 events during your 30 second sample window, switch to average sample rate with a target sample rate of 100.

**The advantage of combining different methods together** is that you can mitigate each of their disadvantages. In this example, the combination lets you keep full detail when you have the capacity, and gradually drop more of your traffic as volume grows.

### Sampling allows for observability at scale

When operating at the scale of hundreds-of-billions of events per year, sampling becomes the only reasonable way to keep high-value, contextually aware information about your service. As your service workload increases, you'll find yourself sampling at 100⁄1, 1000⁄1, 50000⁄1. At these volumes, by using the statistical sampling methods described, you can be sure that any problem will eventually make it through your sample selection. Using a dynamic sampling method will make sure the odds are in your favor.

## Developing an observability-centric culture

A shift in the culture of the software engineering profession is occurring. As the size and sprawl and complexity of our systems skyrocket, many of us are finding that the most valuable skill sets sit at the intersection of two or more disciplines. To translate across these disciplines, we need empathy.

Even a rudimentary grasp of an adjacent skill makes you a much more effective engineer—whether that skill is in databases, design, operations, etc., having it makes you easier to work with. It means you share a common language with your coworkers. It also helps you instinctively avoid making whole categories of mistakes, and it means you'll have the ability to explain things about one side to someone on the other side—in either direction.

Consider the kind of engineers that used to be elevated within the software industry: Our heroes were grumpy engineers who sat in the corner coding all day and speaking to no one. They were proud when people were afraid to approach them and suffered no fools. Your database administrator wanted to be the only one to swoop in and save the day when the database was on fire. Software engineers loved to grumble about how "hard, mysterious, trival, or not worth my time" all the other business lines in the company were, from Marketing to Ops.

But we're learning the hard way just how fragile our systems are with that kind of engineer building and operating the code. We're realizing we need to change the engineering ideal—away from brusque lone heroes with grumpy attitudes and toward approachable, empathetic people who share a generalized tool suite.

## Suggestions for building observability into your culture

- Treat owning your code and instrumenting it thoroughly as a first-class responsibility and as a requirement for shipping it to production.
- Share your learnings, build runbooks, how-to's, and lists of useful queries.
- Find tools that let you do that easily.
- Make it easy for new folks to dig in and come up to speed by building the context they need into your events and tools.
- Incentivize documentation within your tools so you can bring everyone on your team up to the level of the best debugger.
- Value exploration and curiosity, and the discovery of connections and correlations.
- Establish an "observability protocol" based on the answers you gave to the questions in the section on Observability-Driven Development so you don't have to worry about safety and can just focus on performing at your best (much like a harness on a trapeze, it seems constraining but is actually freeing).
- Incentivize the collection of lots of context—immediately remove anything that puts pressure on engineers to collect less detail or select only a limited set of attributes to index or group by.

For more suggestions, take a look at our guide on Developing a Culture of Observability.

# Where might observability take us?

The future is software engineering process–centered, social, focused on the health of the individual request / customer experience, event-driven, richly instrumented, sampled at scale, interactive, and dedicated to turning the grueling mysteries of dealing with unknown-unknowns into simple support requests.

Observability started as a practice once reserved for only some of the world's elite companies (Google, Twitter, Facebook, Netflix, etc.) that were trying to solve seemingly impossible problems. Honeycomb's mission has been to democratize the kind of expertise that having these sophisticated tools unlocks. That's why we brought the first commercial observability solution to market. It's why we're at the forefront of developing observability practices, and it's why we see not just our customers but the entire software industry now buzzing about the importance of observability.

We're seeing the shift toward observability in the form of being able to ask new questions, discovering deeply buried problems, empowering developers to own their services, and adjusting toward having a better understanding of the complex distributed systems we all manage now. As a result, we're also seeing that the ability to easily do real-time, ad-hoc, interactive analyses is changing not just our tools and playbooks, but our cultures as well. As we all learn how to overcome the challenges of working with containers, microservices, various SaaS components, distributed abstracted infrastructure, and the chorus of orchestration services to manage all that, our technology is driving a wave of changes we're all on board to experience together.

Welcome to the world of observability. We welcome you along for the ride.

honeycomb.io

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.