Observability
For Developers

This guide is part of an ongoing series on observability for engineers and operators of distributed systems. We created Honeycomb to deliver the best observability to your team so you can ship code more quickly and with greater confidence.

www.honeycomb.io

We post frequently about topics related to observability, software engineering, and how to build, manage, and observe complex infrastructures in the modern world of microservices, containers, and serverless systems on our blog:

https://www.honeycomb.io/observability-blog/

This is the second guide in our highly-acclaimed observability series. Get the first guide here

Observability for Developers

Introduction

Observability is just as important as unit tests, and operational skills – debugging between services and components, degrading gracefully, writing maintainable code and valuing simplicity – are becoming non-negotiable for senior software engineers, including mobile and front-end engineers. This means software engineers must be comfortable and confident breaking systems, understanding them, experimenting, and fixing them.

Instead of being scared to break things, observability helps you become confident enough to ship more often--because your ability to find and solve problems becomes supercharged. And as you feed the understanding and insight observability garners you back into your development process, you make better decisions about what to build, and when to build it, as well.

What is this guide about?

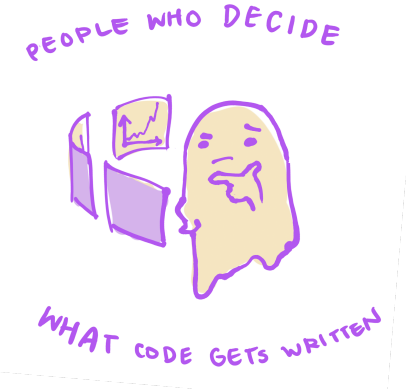
This guide provides the means to measure your current level of observability practice and how to get to the next level. We'll go into detail about not just why, but how you can implement observability as an integral part of your development process and culture.

Looking for an introduction to the concept of observability? Our previous guide is a good place to start:

Who is this guide for?

People who write code–because observability means having the power to not only understand but resolve and improve the experience of your users. Putting that power in the hands of the people who actually write the code means you can build a culture of software ownership.

People who manage people who write code–because observability is more than a technical practice, it is a culture of ownership and control, consistency and confidence. If you make decisions about workflow and process, about how engineers do their jobs and what their responsibilities are when code rolls to production, you should understand the benefits and goals of an observability-driven development practice.



People who decide what code should be written–because observability gives you tremendous visibility into how your users experience your product or service, you can leverage these valuable insights and data to help you prioritize decisions about what features or improvements should be worked on next.

What observability means for product development organizations

Historically, the observation of systems in production has been considered the responsibility of operations organizations--systems administrators and operators, monitoring the health of the overall system via dashboards and alerts.

These systems are now becoming more complex. The nature of delivery has changed from monolithic codebases hosted on company-owned hardware to more loosely-coupled, service-oriented offerings. This has led to an ongoing shift in the delivery and management of services, giving rise to DevOps practices. Understanding the underlying code being run in production is becoming more and more difficult.

The experience of the individual customer is more critical than ever, with users able to share their experiences of a given service with the world via social media.

What has come to the forefront as a result is the need for developers to have deeper visibility into what their code is doing in production, a stronger connection to the end user experience, and how production issues affect individual users of the service. What is needed is observability. But where to start? And what to prioritize?

To begin answering these questions, it's not just desirable, but necessary for product development teams to understand more about the higher-level goals of their deliverables than has historically been the case. Product teams need to understand what they are delivering and why, so developers can use that information to build the right instrumentation into their code. In turn, product owners need the output of instrumentation to make decisions around improving product delivery and development processes.

Observability can help your understanding of what you're actually trying to deliver, improve how you deliver it, and it can also improve your own experience while delivering it. Observability can give you greater confidence in what you're shipping and more freedom to continue to improve the quality of the experience of your users. But you have to start somewhere.

Understanding your goals and priorities

Observability for product development organizations can be broken down into two major, overlapping categories:

User experience

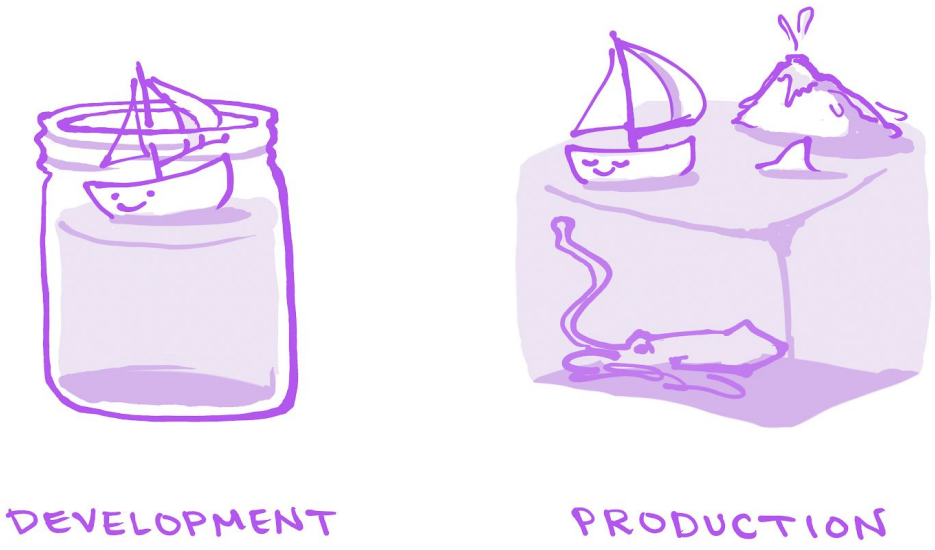
The first thing to determine is: what matters to your users?

In most cases, you're concerned with their experience of your service in terms of performance and stability in both the front and back end, but with the right instrumentation, you can also understand and improve how users experience your UI/UX. Are they using the features in the way you and your Product Management team expected? Are they experiencing frustration?

Observability can tell you these things and a lot more. With this information, you can make better feature and design decisions, address user frustrations quickly and more effectively, and focus on what really matters to your business: user happiness.

Development process

There's a lot of talk lately about "testing in production," and although the term is intended to be somewhat provocative, the truth is that most modern services currently operate at such levels of complexity that it's not really a matter of /if/ you're testing in production--you are.



There are lots of things you already test in production because it's the only way you can test them. You can test subcomponents of varying sizes and types, in various ways and with lots of different edge cases. You can even capture-replay smaller systems or shadow components of production traffic--the gold standards of systems testing. But many systems are too big, complex, and cost-prohibitive to clone. Most have user traffic that's too unpredictable to fake sufficiently well to make a test environment worth the effort it takes to maintain one. Increasing scale and changing traffic patterns are themselves a test of your code, in production.

The reality is, every deploy is a test.

Obviously, we'd all prefer to not test in production, but if you weren't already doing that, you wouldn't need exception tracking. Observability means you can see how your code is behaving immediately after it's deployed. Instrumentation that emits data that developers understand natively brings developers closer to the production environment. When combined with canarying and feature flagging, you gain tremendous control and confidence--observability means developers can much more easily take ownership of their code in production and makes being on call a much less fraught and disruptive experience.

There's a lot of value in testing: to a point. But if you can catch 80-90% of the bugs with 10-20% of the effort by investing a little more in unit testing, the rest is more usefully poured into making your systems resilient and easy to debug, not preventing failure. Prioritize observability.

Evaluate your current observability practices

The questions in this section aren't meant to cover all potential ways in which you can bring observability to your development practice, but they're a good place to start.

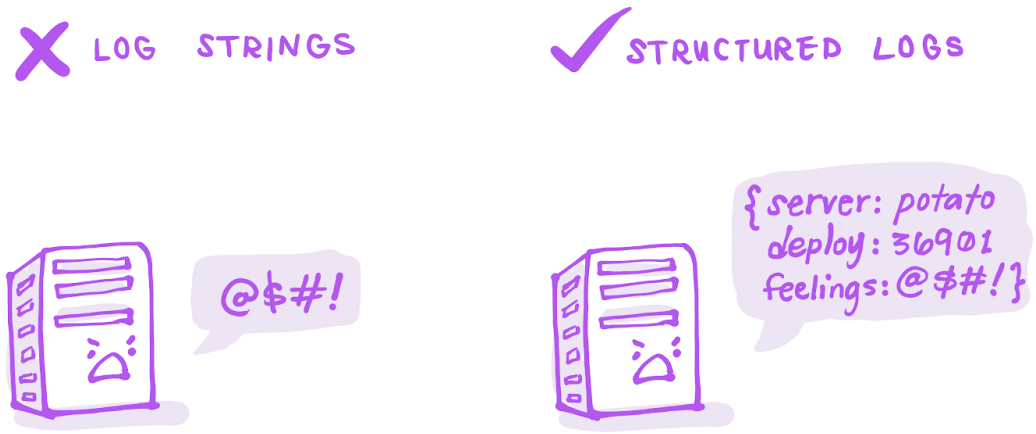
Are your existing logs structured?

Logs are no longer human scale. If your team is trying to function with a logging setup where human brains identify useful pieces of information in variables, embed them in opaque strings, and have computer brains parse them out, observability is not yet within your grasp.

Structured logging is largely about having a logging API to help you provide consistent context in events. An unstructured logger accepts strings. A structured logger accepts a map, hash, or dictionary that describes all the attributes you can think of for an unit of work:

- the function name and line number that the log line came from
- the server's host name
- the application's build ID or git SHA
- information about the client or user issuing a request
- timing information

The format and transport details (whether you choose JSON or something else, whether you log to a file or stdout or straight to a network API) are often less important.



Structured logging is a form of standardization that drives the effectiveness of your team. The consistency allows your team members to level up faster. It's a critical part of joint ownership

and breaking down siloed knowledge. When you are troubleshooting a vague error or symptom, knowing what to look for can be the difference between a confident resolution plan and an anxiety-laden firefight. Some suggestions for structuring your logs:

Don't be afraid to develop your own idioms

It's totally reasonable for a mature project or organization to maintain its own module of logging conveniences. You can have some startup configuration that outputs nice colors and pretty formatting when you're developing locally, but just emits JSON when your service runs in production. You can (and should) have helpers for adding domain-specific context to all your requests to standardize naming schemes and so on (such as customer name, ID, and pricing plan). You can be creative.

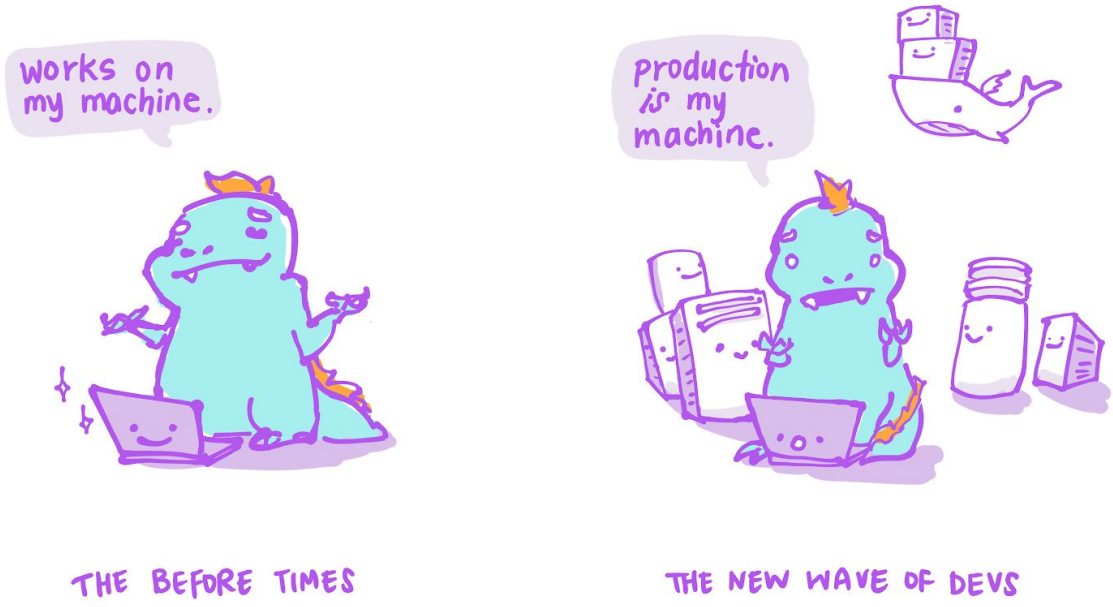
Don't be afraid to make incremental changes

You might have lots of log statements all over your code base. You might have some gnarly logging pipeline. Don't worry about that for now. First, identify your service's core unit of work: is it serving an HTTP request? Is it handling a job in a job queue?

Then, write one structured event for that unit of work. "Every time we're done handling a request, we record its status code, duration, client ID, and whatever else we can think of." Don't sink weeks into changing all of your logging at once. If structured logs for even a subset of your code base help you better understand what's happening in production (and they probably will), then you can invest in standardization. Don't be daunted.

Is instrumentation a core part of your development workflow?

Instrumentation (and the resultant observability) are the new testing. For most teams, tests have become second nature. You already understand the need to validate what you expect to happen against what actually does when your code is run. Observability and instrumentation are how to achieve that when your code is running outside your test environment, in production. And the quality of your instrumentation is directly tied to the quality of your observability--in very much the same way that good, appropriate, relevant tests give you a high-quality understanding of the quality of your code.



The days of developers throwing code over the wall to operations to deploy and run should be over. The days of "works on my machine" are definitely over. Engineers need to see how their code really behaves in the wild. Good instrumentation, and the observability it brings is like debug lines in production; it's how you make your tools use the language your developers understand. Make your output speak in terms of account IDs or assets or whatever matters to your business.

Are your developers able to handle being on-call?

Take a look at all the monitoring tools out there. Count how many of them talk about CPU or disk space. How would a developer debug a CPU or disk space issue if they were on call tonight? Ask your developers how they'd manage to connect CPU utilization to code -- or even figure out if it was a deploy/code change that caused it.

Ideally, everyone who has access to production knows how to do a deploy and roll back, or how to get to a known-good state fast. Everyone should know what a normally-operating system looks like, and how to debug basic problems.

Observability is about developers understanding the sensibilities of operations teams without forcing them to do an extra layer of translation to get at the insights they care about. If your code is instrumented with context and terms that developers understand natively, debugging a production issue becomes much easier. Being on-call becomes less stressful.

Do you use data from your service to inform product decisions?

What are users doing with your product? Is it what you expected? With the right instrumentation and a flexible observability platform, you can find out if the feature you just shipped is frustrating your users, and prioritize work to resolve the issue.

What can you learn from their behaviors? Data you collect can tell you whether they're having difficulty seeing the information you're showing them, if they're finding a particular set of controls confusing, if you're using the screen real estate optimally.

Develop useful instrumentation practices

Observability starts with instrumentation--and it continues with instrumentation. As you learn about what you need to collect to answer the questions you need to answer, you improve the data you collect and solve problems more quickly. In many situations, you will add instrumentation and then remove it later, when your theory has been proven or disproven. Not all instrumentation is cumulative.

Some instrumentation tasks can seem like drudgery, such as adding timing information for various requests and processes--but you can get much of that part done automatically using automatic instrumentation/tracing features like those in Honeycomb Beelines.

Be judicious

Don’t instrument to catch every possible minor failure. In this sense, instrumentation is a lot like unit testing: you don't write unit tests for every single case, you pick a few representative test cases to sanity check your logic. Instrument to capture the information you'll need to drill down later, but capture the pieces of metadata that describe major forks in the road of the logic or behavior that might be worth digging into.

Compare capturing useful metadata to capturing the "partition keys" of your traffic: the most useful bits to capture are the ones that let you rule out large swaths of traffic as irrelevant or uninteresting today. Unsurprisingly, these are often the domain-specific things that APM tools can't just guess for you. They're tied to your custom logic and only you know best what kinds of questions you'll want to answer--but there are some specific recommendations later in this guide.

Log with intent

Teach your code to communicate with your engineers. The events with business value for your organization don't have to be buried in uncontrolled, messy log files anymore. Emitting well-formed (preferably JSON) events for downstream analysis should be a first-class requirement for new applications (and a high priority requirement for refactoring old ones).

Instrument your code to emit data that will be meaningful and potentially actionable. Not the "goto here"-type printfs that made sense in dev, but instead, log with the intent that someone else reading it--possibly even future you--will understand what happened (such as the function that was run) and what caused it (the parameters). In some ways, logging with intent is like commenting or documentation--the form factor and delivery mechanisms are just different.

Search- and index-based logging tools are irrelevant now for debugging and solving production issues. Put the events that the business cares about into a datastore that is structured to support fast, ad-hoc querying on known dimensions and specifically optimized for behavioral queries.

Develop a strong culture of instrumentation

Work with your team to conform to a set of rules, supporting a strong standard of consistency that helps them understand what is going on with any service in your production environment. For example, at a minimum:

- Each request hitting a server is logged at least once
- Each request being handled by the server gets assigned a request ID that accompanies all its logs

- Each response sent by the server is logged with:
    - what type of request was served and
    - for whom, in whatever terms make sense for your business
    - a duration, how long the request was being processed by the server

With this kind of instrumentation, you can answer the following kinds of questions and many more:

- How many requests have been served in the last 24 hours?
- What are the 10 most frequent errors occurring for a specific request type?
- What are the 99th, 95th and median percentiles for duration of request handling by this service?
- Does the service's error rate correlate with time-of-day? Count of requests served?
- If a support request came in from a user stating that they experienced a service outage at approximately 2pm today, what errors occurred during that time? Can we identify the initial request made by that user, and trace it through all of our services to understand what went wrong?

Onboard new team members into the culture of observability

Consider having new hires spend a sprint getting to know the ins and outs of your logging and instrumentation standards, systems, and codebase. Knowing that all services speak the same language adds a very effective angle to this onboarding. New team members learn the system not just from the code and documentation, but from observing the production environment.

Create a set of dashboards and alerts that tell the story of what matters to your team, that showcase the experience of your users and what affects that, so new team members get up to speed while understanding the actual goals of your business. **Show you value ownership of code** by requiring developers of code to be present when it ships.

Instrument at code creation time and link it to deploys

Instrument alongside code creation, (and again, and again as you discover the data you need to find out what's happening in various cases when your users surprise you.) And although we often think of it as drudgery, the act of instrumentation can itself be educational when undertaking it at some point after writing your code. It's a reason to do another pass over your code, with an eye for "what might be interesting?" — a lens that isn't always active when you're writing the code in the first place.

Know what you expect to see when code is deployed

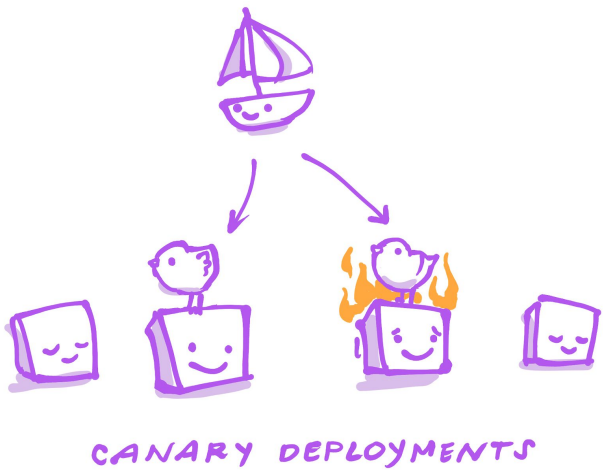
Define what metrics you expect to change (or not change) as the result of a change set ahead of time. Write the instrumentation to validate a change before writing the change code, and **deploy that instrumentation first** so you actually measure the normal behavior in production

before deploying the change code. Don’t fall into the trap of committing code that makes sense when it’s written, but not when it’s run in production. This is one way observability and instrumentation are a bit different from tests; there's an extra step between "write the [test|instrumentation]" wherein you actually run the test/instrumentation. You can run unit tests locally / in CI and it's fine; you must run instrumentation in production to establish the baseline of "what's normal?" before you change it.

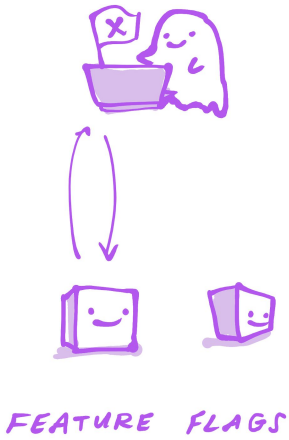
As mentioned earlier, each change set deployed should have a sense of what metric should change or not change as a result of it and that your system overall should have a way to gauge which events are a result of which build/change set. You must have a way of identifying the impact of a single change set / deploy, such as a buildID.

What's the most common source of chaos in systems? Humans. Software engineers. What's the first question folks often ask when confronted with some new degenerate behavior? "When did it start happening?" which is a shortcut to: "When did the code that introduced this land?" on the way to "What code might have caused this?" Short-circuit that series of questions to just: "What code is associated with this degenerate behavior?" directly by tracking Git hashes or build IDs.

Use canaries and/or feature flags



Feature flags and canaries provide further opportunities for you to validate your expectations against reality as well as establish a baseline--and here, the non-canaries or feature-flag-off variants are the baseline. Instead of comparing metrics temporally, you're able to compare metrics side by side from these two groups.



These two approaches are basically the same thing at different resolutions--something in the code vs "this machine is a canary and others are not." If you have well-instrumented code, you can use canaries or feature flags to leverage your observability by comparing the output of your instrumentation and observe the impact of the new code vs the old code. If you use feature flags, you can very carefully vary the breadth of impact of the code while watching the impact--a hugely powerful experience.

Instrument your end-to-end tests

End-to-end testing on its own just tells you whether something failed--which is of course critical. Critical, because it encapsulates the experience of your users in something that is automated and repeatable without a human operator. With the right instrumentation, you can also learn what failure looks like and what it doesn't look like--not just that something failed, but how (how long did it take to fail?) and ultimately why.

Observable end-to-end testing lets you capture what matters to your users about your service, from their perspective, while also capturing the data to immediately debug the service if something goes wrong. You can reliably and succinctly report on the experience of your users this way, and pairing observability with this reporting allows you to investigate quickly when a very high-signal thing reports unexpected behavior, when something outside the normal pattern happens.

Instrument with events and traces

Event based tracing and debugging have become critical for successful distributed systems development and operations.Tracing is a visualization on top of structured events. If you have the attributes that describe order and a way to uniquely identify the items in a given waterfall, you can describe a trace.

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 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.

An event represents a unit of work done by your code. It can tell a story about a complete thing that happened–for example, how long a given request took, or what exact path it took through your systems.

What is actually useful of course depends on the details of your service, but as a point of reference, the Honeycomb front end web server generates events with an average of 40 fields (±20), and our API server is closer to 70 (±30). Building and collecting wide events (with many fields) give you the context you'll need later when trying to understand your production service.

Every time you wire up an event to be dispatched to your logging system, it's a note to your future self. Unfortunately, these aren't stories you read when you're sitting cosily by a fire in a nice fireplace. You read them as you're frantically trying to figure out how your rug caught fire and how to put it out. Generating good events can dramatically improve your ability to respond to system incidents.

What should an event contain?

One of the critical aspects of an event is the unique key that lets you trace sequences of code behavior through a system. A good example of this comes from the world of shipping and warehouses: The unit of work that represents a particular item being selected from a location and placed into a shipping container is typically called a "pick instruction". Pick instructions have a UUID. When you search on that UUID in the inventory system, you find all the things that happened to fulfil that set of instructions from creation to termination (wherein the item was placed in a container). Almost every operation that references or modifies pick instructions will log an event that contains a description of what it is about to do or has done ("calculating inventory assignment", "saved inventory assignment") and the context surrounding it.

The UUID is crucial. It doesn't have to be a "UUID": it can be any unique identifier, a request ID that gets assigned to every HTTP request and passed through your layers–just as long as when you search for it, you will only find events related to it.

General recommendations for event contents

- Add redundant information when there's an enforced unique identifier and a separate column that is easier for the people reading the graph to understand. For example, at Honeycomb, the Team ID is globally unique, and every Team has a name. We add the ID to get a unique breakdown and add the Name so that it's easier to recognize ("honey" is easier to remember than "122").

- 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.
- 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. No big deal.
- Think about your field names some but don't bikeshed (http://bikeshed.com/). Common field name prefixes help when skimming the field list since they're alphabetized.
- Add units to field names, not values (such as parsing_duration_μs or file_size_gb).

Specific recommendations for event contents

This list of fields is applicable to most services you might be running:

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 (such as rails route or goji handler or django view or whatever it's called these days.)
- 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 you 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? etc.)

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?)

What environment did the request occur in or pass through?

- Hostname or container ID
- Build ID
- Environment, role, and additional environment variables
- Attributes of your process, such as amount of memory currently in use, number of threads, age of the process
- Your broader cluster context (for example AWS availability zone, instance type, Kubernetes pod name)

What aspects of your business logic are relevant to this request?

This type of information is often unavailable to each server, but when available it's surprising how useful it can be at empowering different groups to easily use the data you're generating. Some examples:

- Pricing plan – free tier, pro, enterprise
- Specific SLAs – if you have different SLAs for different customers, including that info here can let you issue queries that take it in to account
- Account rep, business unit.
- Additional context about your service / process / environment

How are traces related to events?

At their core, traces are series of events tied together by a common traceID. Tracing makes it easier to understand control flow within a distributed system. Waterfall diagrams capture the execution history of individual requests, making it easy to answer questions such as:

- What does the high-level structure of the code look like?
- Which methods are especially slow?
- Are we making too many calls to a particular service?
- Are things happening serially when they could be parallelized?
- Is there unaccounted-for time where instrumentation is missing?

With traces constructed of events, you can answer questions like:

- Which API endpoints are particularly slow or error-prone?
- How did performance change between deploys?
- Which users are responsible for the bulk of database load?

Tracing helps pinpoint where failures occur and what causes poor performance, but it's often difficult to figure out where in general the problem is occurring so you can drill in and examine the relevant traces. Having access to events that are linked together into traces makes it possible to start with a query across a wide time period, identify an anomaly, and then scope your debugging down to a particular API call at a particular time.

Instrument for user happiness

Observability has a great deal to offer web developers of all stripes, even those who only write front-end code. If anything, front-end developers have the *most* need to understand how their code is behaving in production because their code is so close to users. They are building the surface that customers touch directly, and server logs won't be enough for them to understand whether it's working as designed.

Good front end instrumentation will reveal where your frontend code has errors or bugs, whether users understand your interface and are engaging with it as designed, and whether they're enjoying a snappy, high-performance experience or a sluggish one. **There's a lot more to happy users than uptime.** It doesn't matter how fast your infrastructure is if you haven't found a way to pass that speed onto your users or they can't figure out how to use your software. The right frontend instrumentation should give you the information you need to find out if you're meeting your performance and usability goals.

As Honeycomb Cofounder and CEO Charity Majors says:

> *"Nines don't matter if users aren't happy."*

Front end instrumentation recommendations

In general, record one event per request from the client side. It will look similar to backend events for HTTP requests, but will have slightly different fields.

Basic info about the request

- Timestamp
- Method used (ajax vs. full page load; GET vs POST)
- URI & route
- HTTP status response
- Response content size

User context

- ID of the user (email address, userID)
- A/B test group membership

Performance metrics

- Page load timings using the Navigation Timing API (some great detail on how to do this in this blog post)
- Relevant User Timings & Resource Timings

Environment info

- Build number
- Availability zone
- Browser user agent (browser, device & operating system)
- Memory stats
- Window size, window width
- Screen size, screen width

Go further

If you can query it with JavaScript, you can add it to your events. Collect information to help you decide what to build into your UI, understand what your user devices have available to them and how they experience what you're sending them. For example:

- Fonts, screen dimensions, and color depth
- Browser language
- Online/offline status - know if your users are dealing with connection trouble
- What kind of connection are they using? (DSL, fiber, 3g/4g?)
- Page visibility--is your page backgrounded?

Send an event every time the user does something interesting:

- On page load
- On single page app navigation (like it's a real page load)
- On significant user action (like using a new feature--instrument new features temporarily to know when people use them and what happens)
- On errors (obviously, and of course also send to error tracking tools)
- On page unload (change from what happened at page load, how long the user viewed the page (total elapsed time))

Suggestion: combine page load/unload with significant user actions into a trace.

Find more states that are error-like, and interesting to instrument

- Refresh tracking: Users often try to reload the page because they've gotten into a weird state--CTRL+R can mean that the user is having a bad experience

- Rage click tracking: find out when user clicks a button repeatedly to try and get something to work--javascript that detects multiple clicks on a target within a short period. (See Philip Tellis & Nic Jansma's talk for more https://www.youtube.com/watch?v=dbAise49tWY)

Front-end developers have so much to gain with access to real data and context for how users are experiencing your service. Use observability to look ahead and see the future so you can plan and prioritize UX and functionality improvements based on actual user behaviors.

Deciding what data to keep

Systems tend to generate two types of data: auditable, and operational. How you collect and manage them will differ depending on what your use case is for the data.

Examples of auditable data: transaction logs, replication logs, billing/finance events

With auditable data, the goal is to retain every record. As a result, you must be disciplined in what your logging systems accept. Establish a strict schema, compact rows, predictable layout. There should be some process around changing schema for this type of data to ensure that the data is always arriving and being stored as expected.

Your business should always be able to assume that the dataset is effectively complete when querying your payment history or MySQL replica, and that the shape of it stays highly predictable so you can forecast costs. You need to keep everything, and ensure it stays accessible, for as long as your requirements state.

Examples of operational data: Telemetry, metrics, and context for requests and system components

Operational data is much messier and broader--it can and should contain any piece of information that would make it easier to debug a problem. Developers should not be discouraged from adding fields to events and including the context they know is needed to observe the behavior of their code in production.

You cannot, and should not expect to keep all your operational data as part of your observability practice.

You're never looking for "one event" in an operational dataset. You're looking for a manifestation of a bug, or a particular use case or pattern, or behavior that matches a specific report. Common things are common, so keep a small percentage of them as representatives. Rare things are rare, so keep all of them. Everything exists on a spectrum.

Sampling is necessary

The best approach to handling high volume traffic is through sampling. As services scale, you want to continue to be able to inspect events with rich detail and context. Sampling is the idea that you can select a few elements from a large collection and learn about the entire collection by looking at them closely. By recording information about a representative subset of requests flowing through a system, you can learn about the overall performance of the system. To handle a high volume of traffic, send representative samples of events, and scale and configure sampling according to the value of the events being processed.

Of critical importance is the way you choose your representative set (the sample set) which can greatly influence the accuracy of your results. There's an extensive discussion of various sampling approaches in our previous guide, "Achieving Observability", the upshot of which is that **dynamic sampling is the gold standard** for the majority of the kind of data and volume. By varying the sample rate based on characteristics of the incoming traffic, you gain tremendous flexibility in how you choose the individual events that will be representative of the entire stream of traffic.

Sampling is not just a good idea, it is necessary for modern observability. It is the only reasonable way to keep high value contextually aware information about your service while still being able to scale to a high volume. As your service increases, you'll find yourself sampling at 100⁄1, 1000⁄1, 50000⁄1. At these volumes, statistics will let you be sure that any problem will eventually make it through your sample selection, and using a dynamic sampling method will make sure the odds are in your favor

Conclusion

Observability is about getting the right information at the right time into the hands of the people who have the ability and responsibility to do the right thing. It's about helping them make better technical and business decisions driven by real data, not guesses, or hunches, or shots in the dark. Time is the most precious resource you have: your time, your engineering team's time, your company's time.

Own your code, own your services, own your availability. Don't build things you don't understand. Observability enables and empowers software engineers to own their code. Make it part of your entire product lifecycle. Use it to close the loop between shipping and moving on to the next piece of code. The more you embed it in your process, the more confidence and freedom you have to build more and greater things for your users.

Thank you for reading this book

We'd love it if you shared this with teammates and friends.

Facebook
Twitter
LinkedIn

❦ ❦ ❦

Or, get started with observability.