Guide: Achieving Observability

Table of Contents

Introduction

Observability has been getting a lot of attention recently. What started out as a fairly obscure technical term from the annals of control theory has been generating buzz of late, because it relates to a set of problems that more and more people are having, and that set of problems isn't well-addressed by our robust and mature ecosystem of monitoring tools and best practices.

This document discusses the history, concept, goals, and approaches to achieving observability in today's software industry, with an eye to the future benefits and potential evolution of the software development practice as a whole.

Why do we need observability? What's different now?

It comes down to one primary fact: software is becoming exponentially more complex.

- On the infrastructure side: a convergence of patterns like microservices, polyglot persistence, containers that continue to decompose monoliths into agile, complex systems.
- On the product side: an explosion of platforms and creative ways for empowering humans to do cool new stuff. Great for users, hard to make using stable, predictable infrastructure components.

Systems complexity is a mixture of inherent and incidental complexity. Inherent: how hard the thing you're trying to do is, and how many moving pieces, how fast it needs to update, the user's expectations. Software complexity is derived from things like: size, modularity, coupling, cohesion, number of code paths, how much control flow exists (cyclomatic complexity), libraries, algorithmic complexity, maintainability, Halstead volume (how much information is in the source, such as variables, functions etc). Infrastructure complexity comes from types of services, ways of communicating between them, exposure to the external network, storage types, security requirements, etc. Age, size, number of maintainers, rate of change, all come into play too.

As recently as 5 years ago, most systems were much simpler. You'd have a classic LAMP stack, maybe one big database, an app tier, web layer and a caching layer, with software load balancing. You could predict most of the failures, and craft a few expert dashboards that addressed 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 unknown unknowns--finding out what was going on and why was generally achievable.

But now, with a platform, or a microservices architecture, or millions of unique users or apps, 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.

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. These new issues are "unknown unknowns," 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.

What is observability?

“Observability” is a term that comes from control theory. 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. The observability and controllability of a system are mathematical duals.”

Observability is about being able to ask arbitrary questions about your environment without—and this is the key part—having to know ahead of time what you wanted to ask.

What makes a system observable

Observability focuses on the development of the application and the rich instrumentation you need, not to poll and monitor it for thresholds or defined health checks, but to ask any arbitrary question about how the software works.

A system is “observable” to the extent that you can explain what is happening on the inside just from observing it on the outside. This includes *not having to add new instrumentation to get your new question answered*. Adding new instrumentation isn't wrong (quite the opposite! Instrument as much as you have to to get the level of observability you need to support your service), but your system isn't *appropriately observable* until you don't have to add more to ask 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, it is about understanding what level of service you need to deliver, and instrumenting so you can meet those requirements.

We'll discuss some specific recommendations for instrumenting for appropriate observability later, as well as more detail about what makes a system observable.

Why are we only starting to build observability tools now?

Until recently, we have not typically had the ability to ask these kinds of open-scoped questions. Answering specific questions has been effectively impossible.

For most of the history of this industry, the reason observability has not been available to us has primarily been storage cost. Now that disk is cheap, we can store what we need to achieve appropriate observability.

The late generations of observability tooling have not been made possible by the discovery of some fantastic new computer science, they have been made possible by cheaper storage costs and made necessary by escalating complexity and feature sets (and architecture decoupling) of the services we observe. (Those escalating complexity trends were also made possible by the cheapening of hardware, so it's Moore's law all the way down.)

In the past, all we could afford to look at and care about was the health of a system overall. And we bundled all our complexity into a monolith, to which we could attach a debugger in case of last result. Now we have to hop the network between functions, not just between us and third parties.

The time has come for a more open-ended set of mandates and practices.

Where does monitoring fit in?

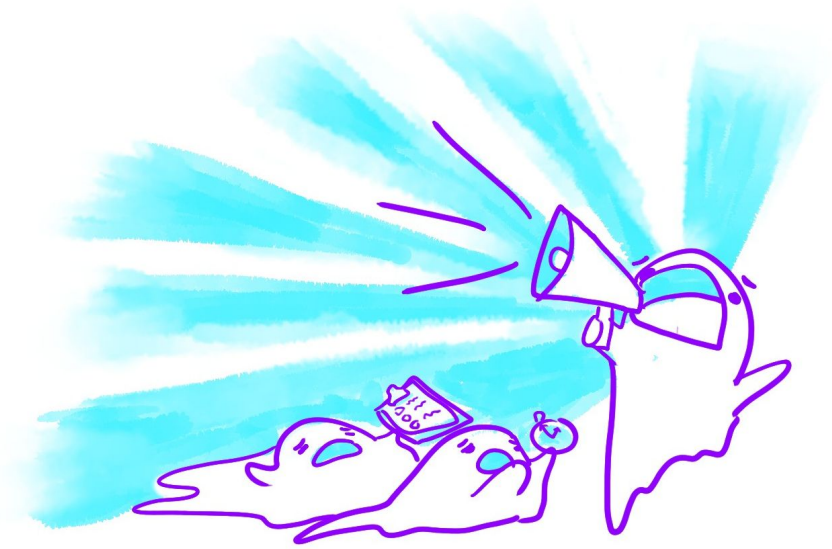
Monitoring, according to the *Oxford Pocket Dictionary of Current English*, is:

"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.

Traditional monitoring relies heavily on predicting how a system may fail and checking for those failures. Traditional graphing involves generating big grids of dashboards that sit on your desktop or your wall, and give you a sense of the health of your system.You can write Nagios checks to verify that a bunch of things are within known good thresholds. You can build dashboards with Graphite or Ganglia to group sets of useful graphs. All of these are terrific tools for understanding the known-unknowns about your system.
But monitoring doesn't help you:

- when you're experiencing a serious problem, but you didn't know for hours, until it trickled up to you from user reports
- when users are complaining, but your dashboards are all green
- when something new happens and you don't know where to start looking



Monitoring doesn’t solve these problems because:

- Monitoring relies on pre-aggregated data.
  Pre-aggregated, or write-time metrics are efficient to store and fast to query, but by pre-aggregating, you are unable to answer any questions you didn’t predict in advance. Monitoring typically relies on metrics, which discard all the context of your events. Metrics are great for describing the state of the system as a whole, but if you throw away everything that correlates to the context of the event, this robs you of your ability to explore, and trace, and answer questions about actual user experience. When a user has a problem, they don’t care about your aggregates, or what the system looked like for most people at that time, or what your 99th percentile was. They care about what *they* are experiencing.
- Monitoring solutions by their nature do not handle querying across high-cardinality dimensions in your data.
  Exploring your systems and looking for common characteristics requires support for high-cardinality fields as a first-order group-by entity.There may be systems where the ability to group by things like user, request ID, shopping cart ID, source IP etc. is not necessary, but if they exist, they are very rare.

Monitoring is good, it will tell you when something you know about but haven't fixed yet happens again. You need that. You also need it to be someone's (everyone's!) job to update, prune, and curate your monitoring dashboards to keep from building up an impenetrable thicket of similarly-named dashboards.

You can definitely learn from rigorously-applied monitoring, and extrapolate to a degree that can mimic access to observability if you're good at making connections and have rapid-fire access to lots of different views into your data.

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

We cannot rely on machine learning or AI yet. It will be a very long time before AI will be able to figure out that...

...on one of 50 microservices, one node is running on degraded hardware, causing every request to take 50 seconds to complete, but without generating a timeout error. This is just one of ten thousand nodes, but disproportionately impacts people looking at older archives.

...Canadian users running a French language pack on a particular version of iPhone hardware are hitting a firmware condition which makes them unable to save local cache, which is why it FEELS like photos are loading slowly.

...the newest SDK makes additional sequential database queries if the developer has enabled an optional feature. Working as intended but not as desired.

Strengthen the human element

A fair bit can be accomplished with rigorous monitoring and automation. With significant commitment to tooling you can keep the on-call paging down to a dull roar, and maybe feel OK about your nines. But you won't be able to keep up without empowering your humans with actual observability.

Instead, we need to strengthen the human element, the curiosity element, the ability to make connections. We need to give people with those skills superpowers, extend their senses, support their intuitions and make it quick and easy to explore a hypothesis, disprove it, and move on to the real root cause.

Who is observability for?

For years, the DevOps community has been iterating on the idea that Operations should do more development. As a result, operations teams now write code, automate their work, and often build their own tools, including observability tools.

Now it's time for a new wave of that movement: for developers to own code in production, and to be responsible for operating and exploring the apps they write as they run in the wild. Observability offers a bridge--a bridge from developers understanding code running on local machines to understanding how it behaves in the wild.

Observability is for developers *and* operators

Observability is all about answering questions about your system using data, and that ability is as valuable for developers as it is for operators. Observability is for software engineers. Observability is for any humans trying to build and run complex, distributed systems. We're all observability engineers.

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 ***highly-cardinal*** data quickly
  When investigating an issue with a production service, you need to be able to get results back in sub-seconds, not minutes.

[sidebar: 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 be 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. Consider: UUID, app name, group name, shopping cart ID, unique request ID, build ID. All incredibly, unbelievably useful. All very high-cardinality.]

- To rapidly iterate through hypothesis after hypothesis and explore the increasingly-various potential causes for the problem

Breadth-first and depth-first investigation options

An event or a trace (traces are a way to look at related events) represents a unit of work in your environment. You need both to fully observe your system.

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 for the benefit of a human who needs as much information as possible, with as much context as possible. At minimum, an event should contain information about the process and host that emitted it, and the time at which it was emitted. Record request, session and user IDs 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.

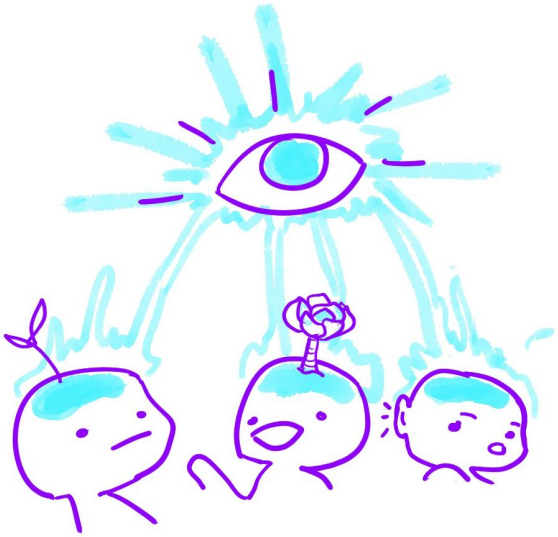
It can be tempting to aggregate, average, or otherwise roll up events into metrics before sending them to save resources, but once you do this you are no longer able to get down to the details you need to truly solve a problem, to ask questions that look more like “business-intelligence” queries.

With wide, context-rich events, when you notice customer experience degrading, you then have a wide 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. (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
- to answer questions without having to re-instrument the service
- to 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 you won’t be able to leverage what you learn, or keep up with the accelerating rate of change and complexity. Choose tools that enhance your team’s ability to collaborate:

- to leverage what the entire team knows about a given problem or service
- to not waste time reinventing the wheel when someone already built a perfectly good bike over there
- to elevate the entire team to the level of the best debugger/problem solver
- to onboard new team members quickly and effectively with real data

Observability-Driven Development practices

Developers who build observable systems understand their software as they write it, include instrumentation when they ship it, then check it regularly to make sure it looks as expected. They use observability:

- to decide how to scope features and fixes
- to verify correctness or completion of changes
- to inform priorities around which features to build and bugs to fix
- to deliver necessary visibility into the behaviors of the code they ship

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 that they can form hypotheses, validate hunches, and make informed decisions—not just when an exception is thrown or a customer complains, but also when decided what to build and how to ship it.

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--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, or 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, you have to start. Don't be paralyzed by what you may think of as how far you have to go, how much work there is to be done. Just start, and tomorrow you will have greater observability than you had today.

Define what you need to achieve

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

[sidebar: 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."]

Understand your priorities

Overall, 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 and 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 the time went.

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.

The health of the system doesn't really matter. Leave that to the metrics and monitoring tools.

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

When you can't instrument the code yourself (for example, the cloud services you use), look for the instrumentation provided and find ways of extracting it to get what you need. For example, for MySQL events, stream events off the wire, heavily sampled, AND tail the slow query log, AND run MySQL command line commands to get InnoDB stats and queue length.

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 which, 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, that collection of log messages, is all related to a single event: the handling of that one connection. But rather than being helpfully grouped into an event, it's spread out into many log messages. Sometimes there's a request ID that lets you put them back together. More often there isn't, and you have to sort through PIDs and remote ports, and sometimes there's just no way to put all the pieces back together. If you can avoid this, you should. If you're in this situation, you should investigate the work required to invest in logging better events.

The definition of an event can be fluid

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 or resolved or discovered along the way, the conditions of the service as it was performing the work, and finally some details on the result of that work.

Treating an event as a unit of work lets you adjust what it means 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, but sometimes 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 hand back a response, but sometimes one HTTP request can generate many events.

The definition of unit of work can change as needed to observe different parts of the service, and the instrumentation you have in that service can change to accommodate those needs. It is a fluid relationship, zooming in on troubled areas, and back out again to understand the overall behavior of the entire service.

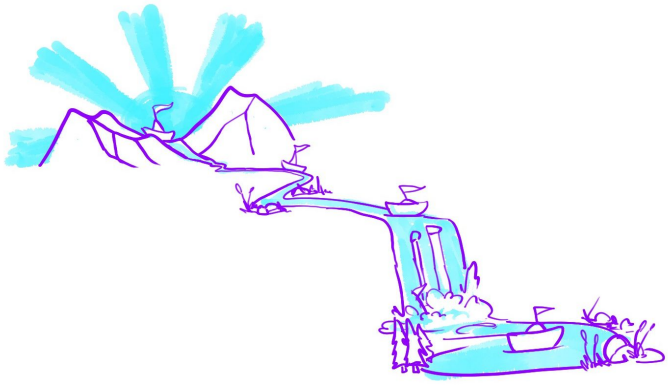
Instrument the code you write yourself

Find a service that offers automatic instrumentation of the basic timings and traces so you can spend your time focusing on the functionality you bring to the table. The following section describes some principles and specific recommendations for evaluating and instrumenting your code.

Instrumenting for observability

What is actually useful is of course dependent on the details of your service, but most services can get something out of these suggestions.

Broad 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).
- 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, 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 of 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.
- 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.
- Be thoughtful about field names.
  Common field name prefixes help when skimming the field list if they're alphabetized.
- Add units to field names, not values (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

Obviously 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:

- Pricing plan – is this a free tier, pro, enterprise? etc.
- 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, etc.

Additional context about your service / process / environment

- Hostname or container ID or …
- Build ID
- Environment, role, and additional environment variables
- Attributes of your process, eg 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.)

Managing the volume of data

Sample your data to control costs, and prevent system degradation, and to encourage thinking of data more holistically in terms of what is important and necessary to collect. All operational data should be treated as though it's sampled and best-effort, as opposed to coming from a billing system in terms of its fidelity. This trains you to think about which parts of your data are actually important, not just important-ish. Curating sample rates is to observability as curating paging alerts is to monitoring — an ongoing work of art that never quite ends.

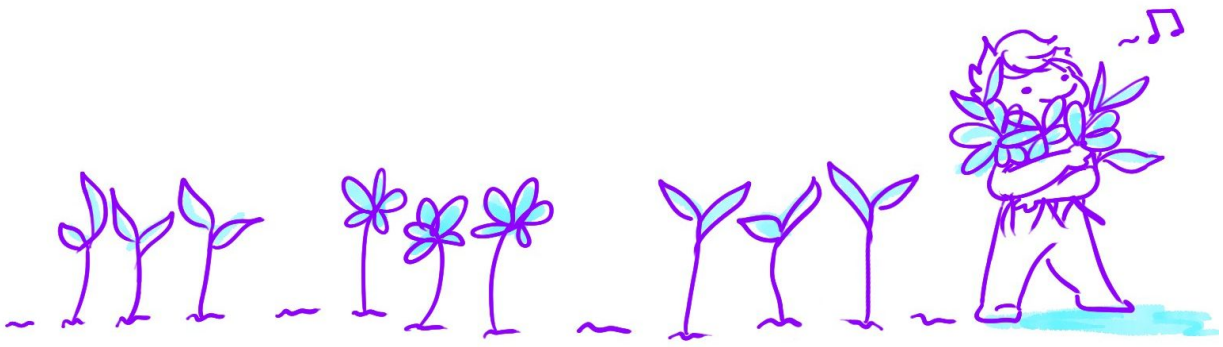
Sampling is the idea that you can select a few elements from a large collection and learn about the entire collection by looking at them closely. It is widely used throughout the world whenever trying to tackle a problem of scale. For example, a survey assumes that by asking a small group of people a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

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

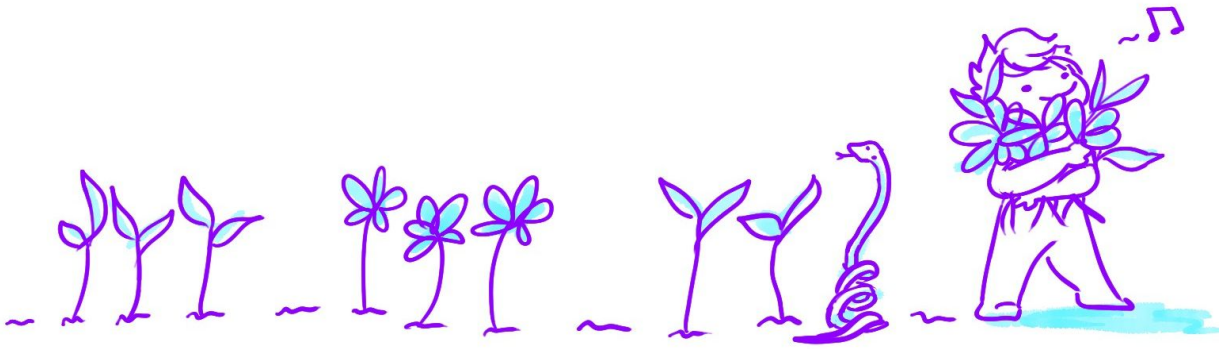
There are two main ways to approach sampling: constant, or dynamic. Dynamic sampling offers greater flexibility.

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 deal with this kind of sampling very easily: multiply all counts or sums by 4. (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, 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 constant. 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

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.

Dynamic sampling: a static map of sample rates

Building a static map of traffic type to sample rate is the the first 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. This way you can represent that you value different types of traffic differently.

By choosing the sample rate based on an aspect of the data that you care about, you gain more flexible control over both the total volume of data you send and the resolution you get.

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, you have a method to manage the volume of data you send in to 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. You must additionally enumerate the details of what makes traffic interesting when creating the map of traffic type to sample rate. While providing more flexibility over a constant rate, if you don't know ahead of time which types might be important, it can be difficult to create the map. If traffic types change their importance over time, this method can not 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, and some types are obviously more interesting for debugging than others. 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 is appropriate 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, so your view

in to 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 in to your data. 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. And 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 (except that it's still happening), but 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 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. Increasing the sample rate on high volume traffic and decreasing it on low volume traffic such that the overall sample rate remains constant provides both– you catch rare events and still get a good picture of the shape 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 is an excellent choice. Picking just one number (the target sample rate) is as easy as constant sampling but you 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.

Average sample rate with minimum per key

This approach combines two previous methods. Choose the sample rate for each key dynamically, and also 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:

- 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.

By combining different methods together, you mitigate each of their disadvantages and keep full detail when you have the capacity, and gradually drop more of your traffic as volume grows.

Sampling is necessary for modern observability

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.

Developing an observability-centric culture

A real shift in the engineering profession is occurring. As the size and sprawl and complexity of our systems skyrockets, many of us are finding that the most valuable skill sets sit at the intersection of two or more disciplines.

Why? Empathy, translation. The ability to intuitively grasp what your cohort is capable of. Even a rudimentary grasp of an adjacent skill makes you phenomenally more powerful as an engineer — whether that is databases, design, operations, etc. It makes you easier to work with, it helps you instinctively avoid making whole categories of stupid mistakes, it means you're able to explain things about one side to someone on the other side — in either direction.

Consider the kind of engineers we used to aspire to be: 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 DBA 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 and Mysterious or Trivial and Not Worth My Time all the other disciplines 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 attitude problems towards 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 learning, build runbooks and 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, 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--a harness on a trapeze seems constraining but is actually freeing; you don't have to worry about safety, you can just perform at your best.
- Incentivize the collection of lots of context--anything that puts pressure on engineers to collect less detail or select only a limited set of attributes to index or group by should be removed immediately.

Where might observability take us?

The future is software engineering process-centered, social, focused on the health of the individual request, event-driven, richly instrumented, heavily sampled, interactive, and dedicated to turning unknown-unknowns into simple support requests.

The systems at Google, Facebook, and Netflix are exponentially more complex than the average tech startup, and the folks behind those systems have to level up before the rest of us. Some practices are still controversial but others have only become more and more entrenched. Advanced internal observability tools start out as tools for specialized use cases like analyzing database performance regressions, then quickly spread to engineering teams across the company.

With the glimpses we get into $BIGCO cultures, we can **see** the shift towards observability: being able to ask new questions, to empower developers to own their services, and to adjust to these systems of increasing complexity.

Many tools developed for internal-big-company use cases aren't quite suited for "real world" stacks — but the ability to do realtime, ad hoc, interactive analysis can change not just our tools and playbooks, but our cultures.

These changes take time to ripple outwards from the companies serving billions of users to the rest of us—but with containers and microservices and the chorus of orchestration services to manage them, our technology is driving a wave of changes we're all on board to experience together. We welcome you along for the ride.