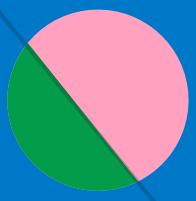
Lightstep
from ServiceNow

# Observability won't replace monitoring (because it shouldn't)

**Ben Sigelman**

*CEO & Co-Founder*



# Introduction

I'm tired of hearing about Observability replacing monitoring. It's not going to, and that's because it shouldn't. Observability will not *replace* monitoring, it will *augment* monitoring.

I'm tired of hearing about metrics, logs and tracing as "the three pillars of Observability." They're hardly pillars—more like "pipes"— and they're the telemetry, not the value.

I'm tired of hearing about vendors and open source software projects claiming to solve all Observability use cases with a time-series database (TSDB). And I'm just as tired of hearing about vendors and open source projects claiming to solve all Observability use cases with an event/trace/transaction store. In fact, to make Observability work, we need both—yet combining them effectively is extremely difficult in practice.

But if Observability won't replace monitoring, if "the three pillars" aren't really pillars, and a single shiny datastore will never be sufficient, how should we model Observability?

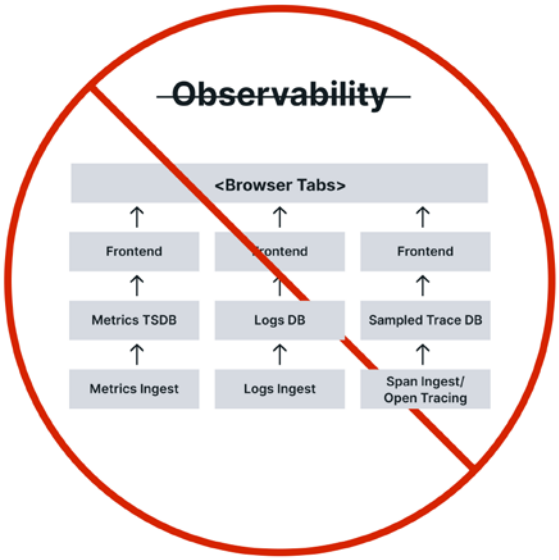
I drew this up and posted it on Twitter a few weeks ago, and it struck a chord:


In this guide, I'll go into more detail and unpack these concepts (more than 280 characters at a time). We'll then understand the proper role—and limitations—of monitoring, the laws of nature governing observability datastores, and how changes should serve as the guideposts for most observability explorations.

Without further ado...

# The anatomy of observability

Before we talk about what the anatomy of Observability *is*, let's talk about what the anatomy of Observability *is not*:

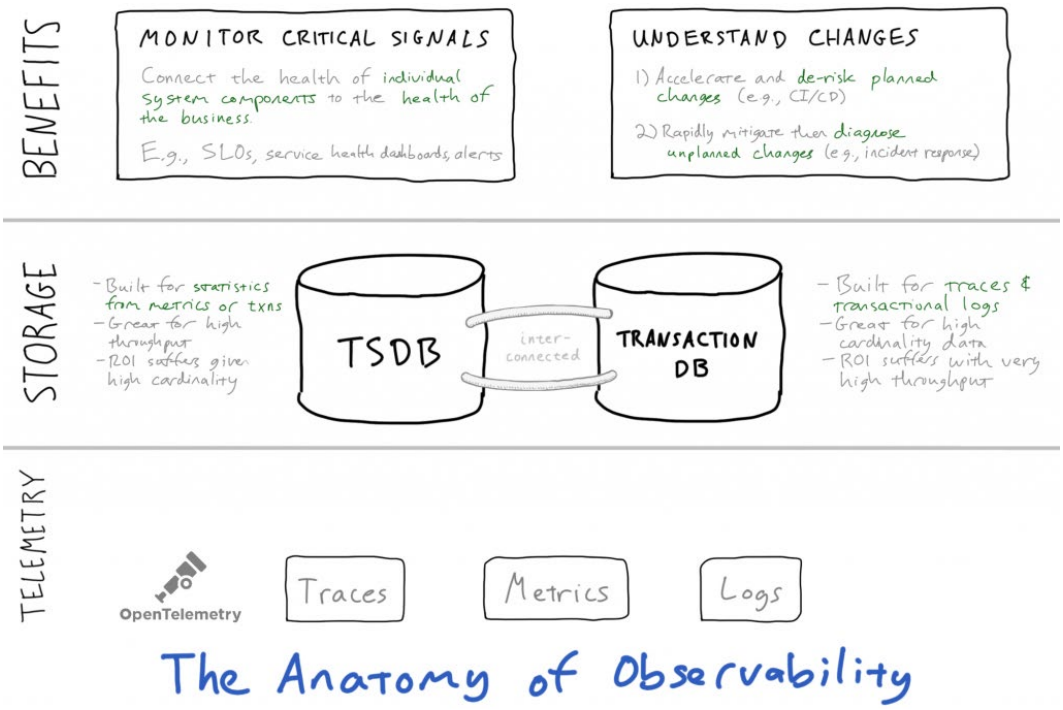



There's been an unfortunate tendency to confuse Observability with Telemetry. Or, at least, with loosely integrated UIs built on top of telemetry silos—as shown in the diagram above. In this formulation, Observability is somehow explained as the mere coexistence of a metrics tool, a logging tool, and a tracing tool. Unsurprisingly, this idea is propped up by vendors who built or acquired products built upon each of those so-called "pillars"—and now they want to sell them to you! Hmm.

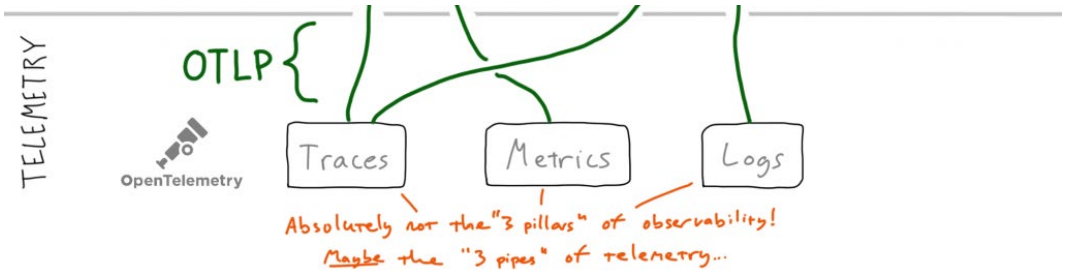
In short: **do not mistake the coexistence of metrics, tracing/APM and logging for "Observability."**

Like most bad ideas that actually gain some momentum, there is a grain of truth here: in particular, that the traces, metrics and logs all have a place in the ultimate solution. But they are not the product—they are just the raw data. The *telemetry*.

So, if Observability is more than "metrics, logs, and traces," what is it?



## Layer 1: (Open)Telemetry




Indeed, we cannot have Observability without the raw telemetry data. As far as gathering that telemetry, historically engineering organizations had two options:

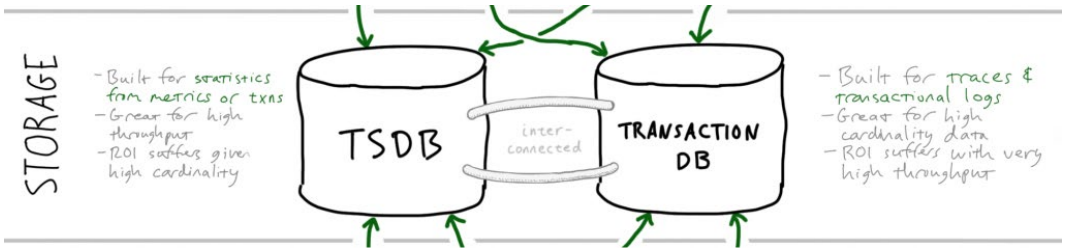
1. Staff up a major, ongoing initiative to gather high-quality telemetry across their infrastructure; or
2. Pay a small fortune to a vendor to deploy proprietary agents that provide easier access to basic telemetry, ideally without any code changes.

Both options were expensive, albeit in different ways—but thankfully there's now an alternative that brings the best of both worlds: OpenTelemetry.

The OpenTelemetry project (aka "OTel") aims "to make high-quality telemetry a built-in feature of cloud-native software." All major cloud vendors have pledged their support for OTel (including Amazon Web Services, Azure, and GCP) and the same can be said for most vendors offering some sort of Observability solution (including Lightstep, as a founding member of OTel, naturally!).

Since OTel supports automatic instrumentation, it's now possible to get high-quality, vendor-neutral telemetry without code changes. In this way, OTel is disrupting the *idea* of proprietary agents and, in turn, the vendors that differentiated with them. Plus, with native OTel and OTLP ("**O**pen**T**e**L**emetry **P**rotocol") support arriving across the infrastructure stack, there will no longer be a need to buy Observability from a vendor just to gain access to their suite of data integrations. Any OTel-compliant Observability provider will have access to high-quality telemetry, across the stack, and all with zero vendor lock-in. It's better for everyone—perhaps with the exception of incumbent vendors, who must make peace with the sunk cost they've spent on innumerable "cloud integrations."

## Layer 2: Storage




If “Layer 1” is about gaining access to high-quality telemetry, then “Layer 2” is about where you *send* it, as well as how—and for how long—you *store* it. When considering datastores for Observability, I’m reminded of a bumper sticker I saw a while back. It read:

> **Create your own deity!**
>
> ☐ Omniscient ☐ Omnipotent ☐ Benevolent
>
> `Pick two.`

When it comes to Observability datastores, we find ourselves in a similar predicament. Rendered as a bumper sticker, it might look like this:

**Create your own observability datastore!**

☐ High-Throughput ☐ High-Cardinality

☐ Durable ☐ Affordable

`Pick three.`

Fundamentally, if we want to handle high-throughput data efficiently (for example, accounting for 100% of all messages passed in a scaled-out app, or even taking high-fidelity infra measurements like CPU load or memory use per container), we must record statistics to a time-series database. Otherwise, we waste too much on the transfer and storage of individual events. And while some might suggest you can sample the events, for low-frequency data hidden within the high-frequency firehose, you can miss it altogether. This situation calls for a dedicated **Time Series DB** (TSDB): a data store designed specifically for the storage, indexing and querying of time-series statistics like these.

And yet! If we want to handle high-cardinality data (for example, per-customer tags, unique ids for ephemeral infrastructure, or URL fragments), a TSDB is an unmitigated disaster—with the explosion of tag cardinality comes an explosion of unique

time series, and with it an explosion of cost. And so there must be a **Transaction DB** as well; traditionally this was a logging database, although it's wiser to build around a distributed-tracing-native Transaction DB (more on this later) that can kill two birds (logs and traces) with one stone.

Still, finding best-of-breed Transaction and Time Series Databases is necessary but not sufficient. To make the actual "Observability" piece seamless, the data layer needs to be integrated and cross-referenced as well—preferably deeply.

No wonder high-quality Observability can feel so elusive. This brings us to the third and most important layer of our Observability Anatomy...

## Layer 3: Actual Benefits

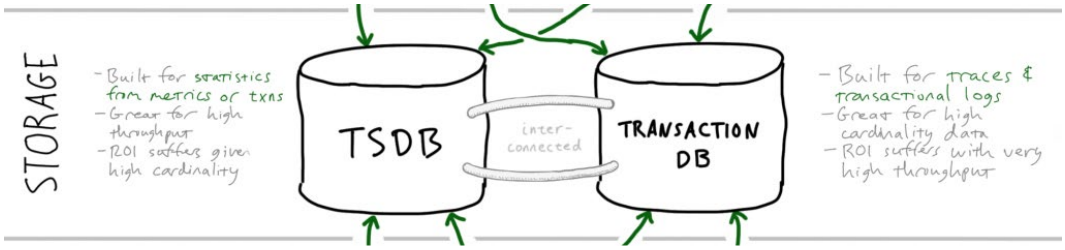
At the end of the day, telemetry—whether in motion or at rest—is not intrinsically valuable. It's only the workflows and applications built on top that can be valuable. Yet in the conventional presentation of "Observability as Metrics, Logs and Traces," we don't even know what problem we're solving! Much less how we're solving it.

When it comes to modern, distributed software applications, there are two overarching problems worth solving with Observability:

- **Understanding Health:** Connecting the well-being of a subsystem back to the goals of the overarching application and business via thoughtful monitoring.

- **Understanding Change:** Accelerating planned changes while mitigating the effects of unplanned changes.

## Understanding Health: “Thoughtful Monitoring”




Somewhere along the way, “monitoring” was thrown under a bus—which is unfortunate. If we define monitoring *as an effort to connect the health of a system component to the health of the business*, it’s actually quite vital. The cleanest and most progressive way to approach this is with a well established organizational process around service level objectives (SLOs), but it’s a sliding scale. What’s most important is that there’s continuous monitoring in place for the key APIs and control surfaces in each subsystem and that an operator can quickly assess subsystem health—at a glance—and quickly spot when and where a particular symptom might have flared up.

For maximum flexibility and minimum redundancy, thoughtful monitoring must be able to measure symptoms found in the Time series DB *or* the Transaction DB. That is, monitoring certainly needs access to the metrics data, but it also needs to be able to query the transaction (i.e. tracing) data in real-time.

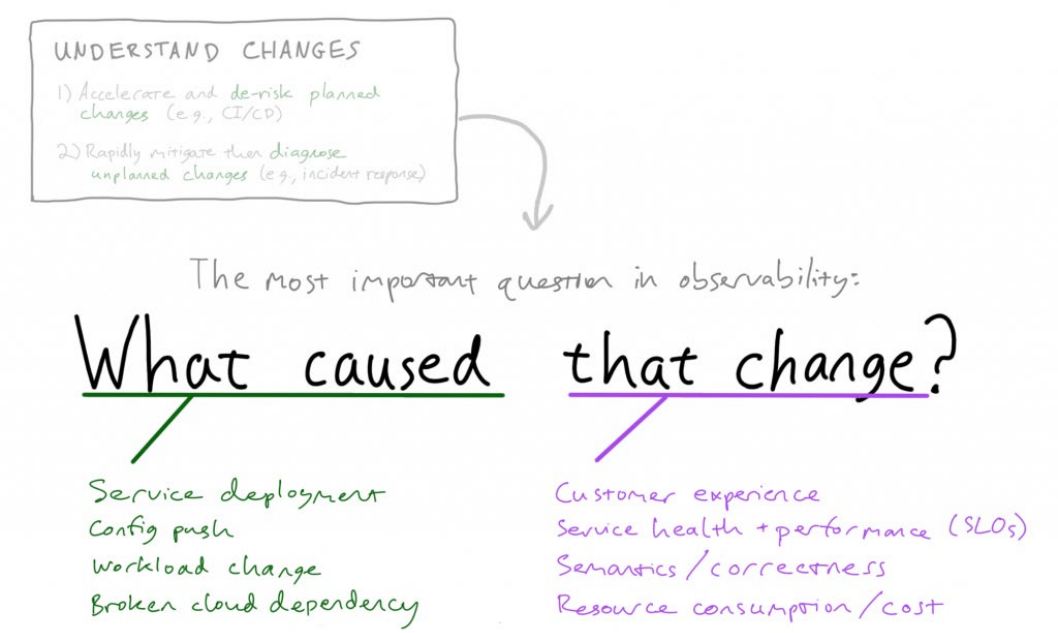
In any case, monitoring should only be used to measure subsystem health and gain advance warning about threats to that health. Monitoring is *not* where we're meant to diagnose changes to that health; and this is where so many organizations have gone astray.

**If you try to solve incident *response* with the same basic monitoring tooling you're using for incident *detection*, you're in for a world of pain.** That's because dashboards and alerts—and even SLOs—are great for detecting health problems, but lousy for diagnosing or even remediating them.

"Monitoring" got a bad name because operators were *trying to monitor every possible failure mode of a distributed system*. That doesn't work, because there are way too many of them. And that's why you have too many dashboards at your company.

So how do we effectively isolate and identify these failure modes?

## Understanding Change




If we only needed to stabilize our software, there's no question where to start: for cloud native apps, more than 70% of all incidents stem from some sort of intentional change (typically a service deployment or a config push). Unfortunately, if we stop pushing new code to production, our products lose out to competitors and our employers eventually cease to exist! So that can't be the long-term strategy, even if we'll still want to declare deploy freezes over holiday weekends.

Stepping back, there are really only two types of change: *planned* and *unplanned*. For *planned* changes, Observability can close the loop for CI/CD and give us more confidence about the local and global health of our deployments (more about that here and here).

And for *unplanned* changes, Observability is the only game in town: since distributed applications have uncountably large numbers of potential failure modes, our approach to quickly understanding unplanned changes must be dynamic and *must* cross service boundaries. At the outset, this means that our workflow *must* touch both the time series metrics that often start these investigations, as well as the tracing data that guides our analysis from one service to the next.

Some have tried to tackle this massive data analysis problem with a similarly massive query language and lots of detailed documentation. This doesn't work, for organizational reasons: the problem is that **Observability becomes an expert system** and in the process, it becomes inaccessible to the generalist engineer/DevOps/SRE who's wrestling with the "unplanned change" we started with.

Let's state the obvious: **Observability isn't valuable if you need to retrain your organization in order to leverage it**. Observability *can* be transformative for an organization, but only if its power is self-evident and accessible. And nobody likes reading manuals, particularly during an incident!

This is why "change" needs to be a fundamental primitive for effective, well-adopted observability. When Observability is framed in the context of a known change, the insights buried in the firehose of telemetry data can be contextualized and *ranked*. In this way, the focus on "change" is mutualistic for both the operator and the tooling: it gives the tooling a clear objective function to explain, and it gives the operator a more digestible UX.

The whole point of Change Intelligence is to start with an identified change—perhaps from an alert, perhaps from a CI/CD system, perhaps just from clicking on a deviation in a dashboard —and *have the Observability solution explain the change*, all without meddling with manual queries or thinking about where the telemetry came from or where it's stored.

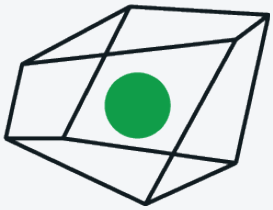
# In Conclusion

Observability seems important because it *is* important. But just because it's important doesn't mean that it needs to feel complex; and it certainly doesn't need to feel confusing.

So let's boil the above down to four short guidelines:

1. Adopt an open, portable means of gathering the raw data (the "telemetry")—with OpenTelemetry being the clear default.
2. Storing that telemetry is fundamentally complicated. Beware any solution that claims to do so with *only* event data or only time-series statistics—both are needed at scale.
3. Monitoring isn't going away, because understanding the health of your system is always going to be important. But Monitoring needs to evolve to connect the needs of the business with the behavior of the application, as directly as possible.
4. And finally: Observability is most accessible, actionable and broadly applicable when it is anchored *around* changes. Either the planned changes of software deployments or the unplanned changes of incident response.

With this framework, we see how to integrate telemetry, how to sniff out a data layer that's too good to be true, and how to evolve conventional monitoring while integrating the valuable insights of a modern observability solution.



## About Lightstep

Lightstep's observability platform is the easiest way for developers and SREs to monitor health and respond to changes in cloud-native applications. Powered by cutting-edge distributed tracing and a groundbreaking metrics database, and built by the team that launched observability at Google, Lightstep's Change Intelligence provides actionable insights to help teams answer the question "What caused that change? Lightstep empowers reliability and observability managers to minimize the impact of outages, improve software performance, and build org-wide standards to drive adoption and automation. For more information, visit Lightstep.com or follow @LightstepHQ.

## Get Lightstep for free

Sign up for our free Community plan, drop by our office hours, or schedule a demo with our team.