Lightstep

The complete guide to observability

James Burns

Lightstep Head of Research

About James Burns

From network load balancers to FPGAs to ASICs to embedded security to cloud ops at scale, James has seen how systems work but, more interestingly, how they fail.

Follow on Twitter
@1mentat

Connect on LinkedIn
james-burns-7816a82/

He is passionate about sharing what he's learned to level up teams, make developers happier, and improve customer experiences. He believes that "if it ain't broke, break it so you know how to fix it," should be the motto of developers, DevOps, and SREs everywhere. While at Twilio and Stitch Fix, James worked directly with engineering teams at large cloud service providers to root cause customer impacting issues quickly.

As the Head of Lightstep Research, he's able to share publicly the tools and techniques that enabled this close collaboration.

Modern systems are complex.

Easy-to-understand (or at least “knowable”) monoliths are giving way to distributed systems: microservices, serverless, meshes, proxies, and every possible combination. These systems enable development teams to build new features and technology faster, as they are not bound to the elaborate release processes associated with most monolithic architectures.

But, like all good things, there is a price: Distributed systems present unique and often difficult operational and maintenance challenges.

When something breaks, it can be difficult to restore service quickly, or even know where to begin. How can you parse through sprawling dependencies—services, databases, versions, cloud providers, managed services, and everything else that could be causing an issue or affected by another?

You can’t attach a debugger to your entire production environment. You can’t rely on stack traces to diagnose distributed issues in a distributed system. There are times when you can’t seem to figure out who is responsible for a particular service, let alone how to repair it.

Adding fuel to the flames, modern systems are expected to be available 24/7, regardless of the pressures placed on the system by peak load, traffic spikes, or any number of variables associated with scaling across devices, geographies, etc.

Which brings us to the question — and the central focus of this guide — how can we effectively operate, build, and manage distributed systems?

We need more than traditional logs and infrastructure metrics in order to manage and understand multi-layered architectures. Thankfully, modern tools allow us to generate, collect, and aggregate high-quality telemetry data, which we can use to observe and determine the state of our system.

All of this, together, forms the practice of observability.

What is observability?

Observability helps developers and operators ("DevOps") understand distributed systems: what's slow, what's broken, and what needs to be done to improve performance.

It's typically defined as a measure of how well you can infer the internal state of a system using only its outputs.

To put this in context, let's look at a real-life analog for observability—cruise control. (Note: We are not, and I repeat not referring to the terrible sequel to Speed.)

Under constant power, a car's velocity would decrease as it drives up a hill. In order to keep the vehicle's speed consistent, regardless of the grade or terrain, an algorithm changes the power output of the engine in response to the measured speed. This system, cruise control, is able to infer the state of the vehicle by observing the measured output. In this case, it's the speed of the car.

For observability, the outputs that matter most are telemetry data—granular measurements of the system that allow Service Level Indicators (SLIs) such as latency, uptime, and error rate to be measured and explained.

Telemetry data: logs, metrics, and traces

There are three primary types of telemetry data through which systems are made observable. We define them in brief below:

Logs

Structured or unstructured lines of text that are emitted by an application in response to some event in the code. Logs are distinct records of “what happened” to or with a specific system attribute at a specific time. They are typically easy to generate, difficult to extract meaning from, and expensive to store.

Structured log example:

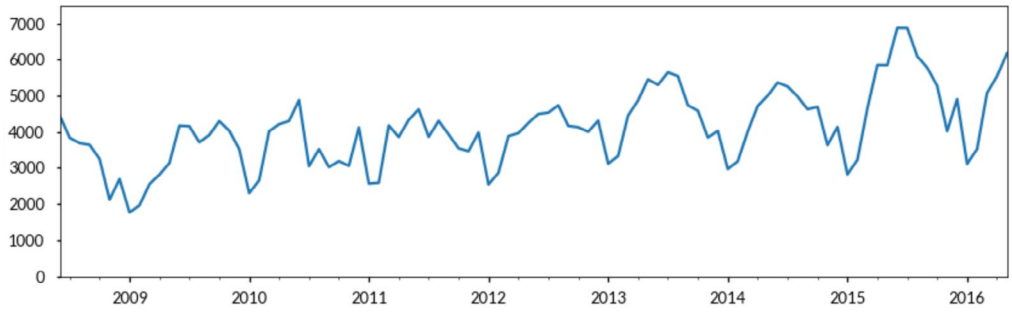
```
127.0.0.1 - frank [10/Oct/2000:13:55:36 -0700] “GET /apache_pb.gif HTTP/1.0” 200 2326
```

Unstructured log example:

```
“thing happened”
```

Metrics

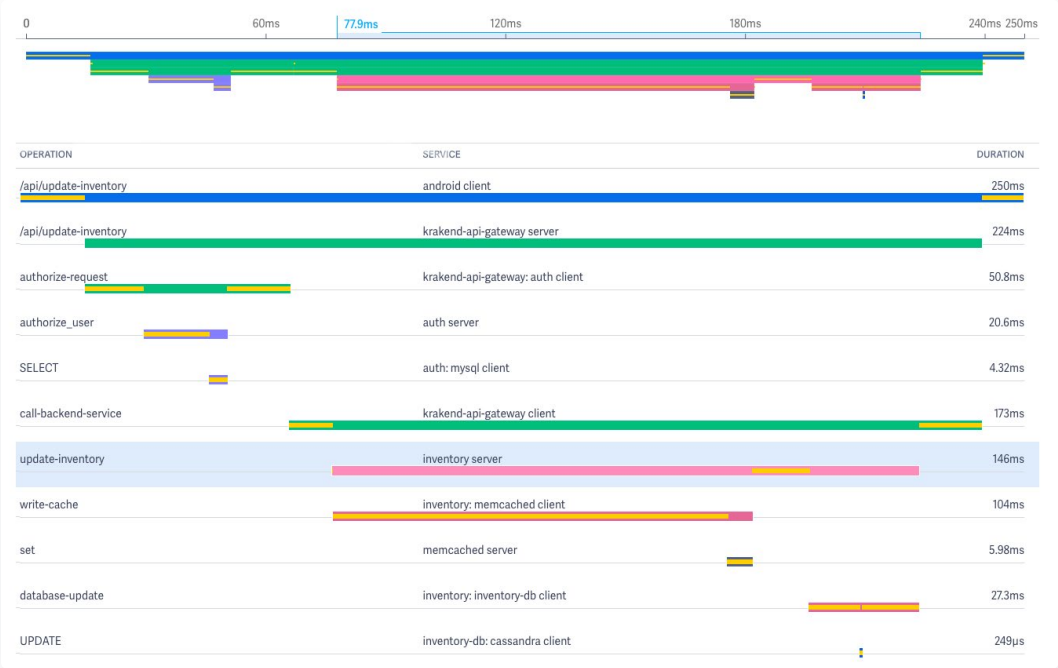
A metric is a value that expresses some data about a system. These metrics are usually represented as counts or measures, and are often aggregated or calculated over a period of time. A metric can tell you how much memory is being used by a process out of the total, or the number of requests per second being handled by a service.



Example of a metric

Traces

A single trace shows the activity for an individual transaction or request as it flows through an application. Traces are a critical part of observability, as they provide context for other telemetry. For example, traces can help define which metrics would be most valuable in a given situation, or which logs are relevant to a particular issue.



The three pillars of observability

Historically, the three types of telemetry have been referred to as the “three pillars” of observability: separate data types often with their own dashboards.

The growing scale and complexity of software has led to changes in this model, however, as practitioners have not only identified the interrelationships between these types of telemetry data, but coordinated workflows involving them.

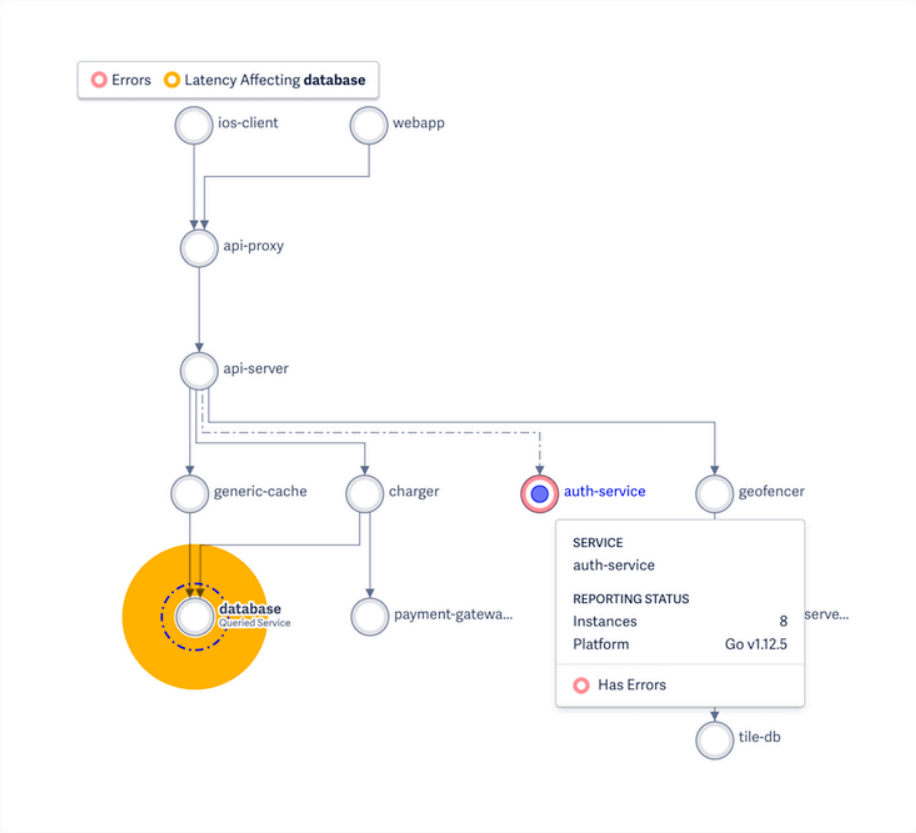
For example, time series metrics dashboards can be used to identify a subset of traces that point to underlying issues or bugs—and log messages associated with those traces can identify the root cause of the issue. Then, new metrics can be configured to more proactively identify similar issues before the next incident.

Also, when viewed in aggregate, traces can reveal immediate insights into what is having the largest impact on performance or customer experience, and surface only the metrics and logs that are relevant to an issue.

Let’s say there is a sudden regression in the performance of a particular backend service, deep in your stack.

It turns out that the underlying issue was that one of your many customers changed their traffic pattern and started sending significantly more complex requests. This would be obvious within

seconds after looking at aggregate trace statistics, though it would have taken days just looking at logs, metrics, or even individual traces on their own.



In short: observability is not simply telemetry—it's how that telemetry is used to solve problems and ultimately create a better experience for customers.

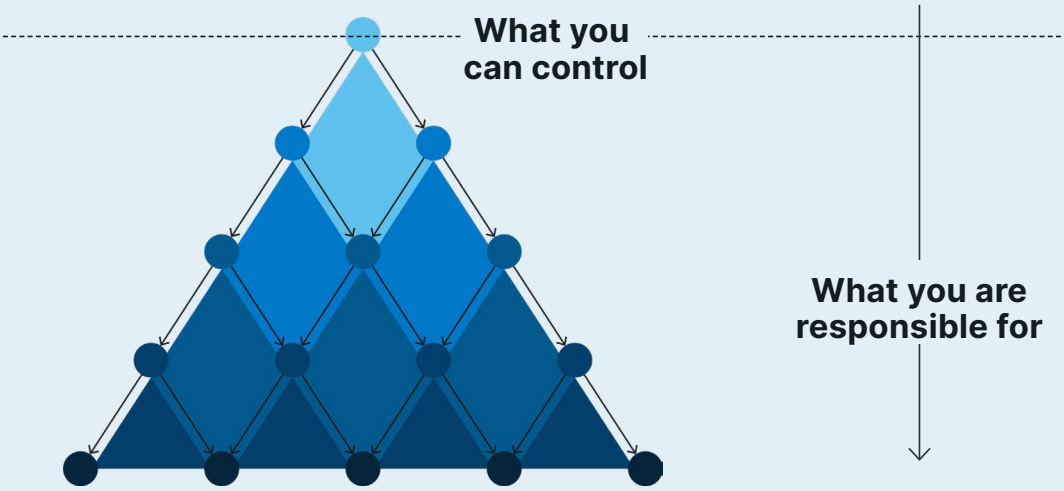
Why distributed systems require observability

In a distributed system, developers are responsible for more than they can possibly control.

Whether they own a single service or (more likely) a group of services, those are invariably tied to and reliant on a handful of other services, which in turn are reliant on other services, ending up with dozens if not hundreds of dependencies: SaaS providers, databases, other services, etc., whose performance can negatively impact their own.

Ben Sigelman, Lightstep CEO and co-founder of OpenTracing and OpenTelemetry, explains this well: “For the human beings who maintain a service, *the scope of control* is really ‘just that service.’ This is by design: the whole *point* of microservices is that individual teams can deploy and operate their own services, all without interference or artificial barriers involving other teams. Put another way, overlapping control was what made monoliths such a disaster for developer productivity and release velocity; hence the very narrow scope of control for microservice owners.

The *scope of responsibility*, however, is everything beneath the given service. Service-owners are responsible for their service’s latency and error SLOs, but their service can only be as fast and reliable as its slowest, least-stable dependency.”



How does this look in practice?

Let's say you build, monitor, and maintain Service A. If Service A depends on Service Z—perhaps through several intermediaries—and Service Z pushes a bad release, that will very likely affect the performance and reliability of Service A and everything in between.

"For the human beings who maintain a service, the scope of control is really 'just that service.'

Ben Sigelman, Lightstep CEO

An observability solution can find extremely strong statistical evidence that the regression in Service A's behavior is due to the change in the version tag in Service Z, and can correlate the negative change to other metrics and logs in Service Z—both before and during the bad release.

Without observability, it would be virtually impossible to gain control over this system on a timeline that would be acceptable for customers. Conversely, having this control enables developers in distributed systems to ship code faster, as they are able to quickly (or automatically) roll back problematic deployments and identify performance issues quickly and accurately.

What questions can observability answer?

This is undoubtedly an incomplete list, but given that the application of observability is so broad, we thought it'd be helpful to provide a sampling of possible questions that can be addressed by an effective observability solution:

- Why is x broken?
- What services does my service depend on — and what services are dependent on my service?
- What went wrong during this release?
- Why has performance degraded over the past quarter?"
- Why did my pager just go off?
- What changed? Why?
- What logs should we look at right now?
- Should we roll back this canary?

- Is this issue affecting certain Android users or all of them?
- What is system performance like for our most important customers?
- What SLO should we set?
- Are we out of SLO?
- What did my service look like at time point x?
- What was the relationship between my service and x at time point y?
- What was the relationship of attributed across the system before we deployed? What's it like now?
- What is most likely contributing to latency right now? What is most likely not?
- Are these performance optimization on the critical path?

Managing observability: SLAs, SLOs, and SLIs

When customers are unable to log into your app or use your product, it's clear that something is not working as it should be. But between "not working" and "working well for every user" are what amount to infinite states of health—for specific services, for system performance, and for how people interact with your product.

This is what SLAs, SLOs, and SLIs are for: to create agreed-upon measurements for service health, system reliability, and consumer or end-user experience.

SLIs

SLIs are Service Level Indicators. These are measurable data such as latency, uptime, and error rate. Through monitoring and observability tools, this data can be tracked and recorded over time.

SLOs

An SLO is a Service Level Objective. It defines the target for SLIs. For example, p99 latency < 1s; 99.9% uptime; <1% errors. etc.

Why does this matter?

SLOs reflect customer expectations. If they are not met, it is likely that customers (or at least a subset of customers) will have a poor experience. Without SLOs in place, agile development breaks down entirely.

A hypothetical: You release a new service, and it has some cool features. You Slack the team, and a few people start using it—and they love it. Your service starts to become popular, and other developers begin to rely on it. Someone on another team finds your service and decides to use it instead of writing their own, but now you're serving more requests than you thought you'd be. Now things are bad. Crashlooping. Texts in the middle of the night. A flurry of angry customer tweets.

A public feature is down because your service broke it.

How would an SLO have fixed this? Well, they're published, for one. Would a developer have started relying on your service if they knew you had a 48-hour turnaround on even looking at issues opened against it? Or if they saw you had an error budget of 10%? SLOs would have ensured that all parties understand what is considered "healthy" for your service.

SLAs

SLA stands for Service Level Agreement, which is typically a financial or contractual consequence for failing to meet SLOs. (All of the five nines jokes go here.)

SLAs are mostly something that you get in trouble for breaking. That trouble might be financial, such as cash penalties or customer

discounts for breaking an SLA, or organizational (i.e., your team gets reorg'ed into deadend project dungeons because you keep messing things up for everyone else).

SLAs aren't really for you, person reading this — they're for people that consume your service.

A good SLA is going to be realistic, but it can be crafted in a way that gives you a lot of latitude on how to actually fix your service or make it better. You can (and should) be specific with SLAs: perhaps there are critical consumers that have them, but others don't. This gives you a knob to turn for performance tuning or request throttling, and allows you to focus on optimizations that have the largest impact to the business.

Also it's important to be able to quantify "legitimate" traffic or requests in your SLA. Maybe a consumer starts sending malformed queries to your service, for which you handle and return an error. Should those count against overall uptime? Probably not, and they arguably shouldn't count against the overall performance of your service.

Without SLIs, SLAs, and SLOs, system reliability is too subjective. There's no clear way to define if your service or system is indeed reliable, or if performance optimizations should be made to increase its reliability.

Benefits of observability

As systems change—either on purpose because you're deploying new software, new configuration, scaling up or down, or because of some other unknown action—observability enables developers to understand when the system starts not to be in the state it's intended to be in.

Observable systems are easier to understand, easier to control, and easier to fix.

Modern observability tools can automatically identify a number of issues and their causes, such as failures caused by routine changes, regressions that only affect specific customers, and downstream errors in services and third-party SaaS.

Benefits to developers

Observability **reduces the amount of stress** when deploying code or making changes to the system. By highlighting "what changed" after any deploy or alerting on p9x outliers, customer-affecting issues can be found quickly and rolled back before SLOs are broken.

With a real-time understanding of full-system dependencies, developers spend **far less time in meetings**. There's no longer a reason to wait around on a call to see who owns a particular service, or what the system looked like hours, days, or months before the most-recent deployment.

By revealing the critical path of end-to-end requests, and surfacing only the relevant data to resolve an issue, observability enables **better workflows** for debugging, performance optimization, and fire fighting.

Ruling out signals that are unlikely to have contributed to the root cause, developers can form and investigate **more effective hypotheses.**

Benefits to the team

For teams of all sizes, observability offers a **shared view of the system**. This includes its health, its architecture, its performance over time, and how requests make their way from frontend / web apps to backend and third-party services.

Observability **provides context across roles and organizations**, as it enables developers, operators, managers, PMs, contractors, and any other approved team members to work with the same views and insights about services, specific customers, SQL queries, etc.

Since observability tools enable automated capture of any moment in time, it serves as a **historical record** of system architecture, dependencies, and service health — both what was and what changed over time.

Observability drives **more effective post-mortems** following incidents, because it allows the team to revisit actual system behavior at the time of the incident, rather than rely on the recollections of individuals operating under a stressful situation.

Benefits to the business

Ultimately, by making a system observable, organizations are able to **release more code, more quickly, and more safely.**

What often determines whether your business is successful or not is the ability to change, and ship new features. But that change is directly opposed to the stability of your systems. And so there's this basic tension: You need to change so that you can expand your business. But whenever change is introduced, risk is introduced, and could create negative outcomes from a business sense.

Observability resolves this basic tension. Businesses can make changes with higher levels of confidence and figure out whether those are or are not having the intended effect, and **limit the negative impact of change**.

The result is more confident releases at a higher velocity. You can deploy more frequently and with greater confidence, because you have tooling that will help you understand what goes wrong, isolate any issues, and make immediate improvements.

Ultimately, it allows you to keep customers happy with less downtime, new features, and faster systems.

Common observability challenges in distributed systems at scale

There are a number of challenges with complex, distributed systems that limit an organization's ability to ship code quickly, resolve performance regressions, and deliver on SLOs. All of these stem from a lack of control (but not responsibility).

Siloed knowledge

Only the developer who wrote a specific service can understand it.

The blame game

Disputes about which service is at fault, when something breaks or is performing poorly.

Shared resources as shared problems

When messaging, queuing, or other multi-tenant systems break and impact multiple services across the system.

Unknown dependencies

Not knowing which services you depend on, and which services depend on you.

Poor experience for VIP customers

Inability to understand the end-user experience for highest-value customers.

Politics and organizational structure

Requests for uptime agreements or other SLOs that are reliant on services many layers down in the stack and outside of your control.

Observability tools: rules that cannot be broken

There are three main ways to make a distributed system observable. Like any choice, each comes with a set of benefits and costs (the latter of which may be literal).

Teams can build their own observability tools, work with open source software such as Jaeger or Zipkin, or purchase an observability solution.

Regardless of how you decide to make your system observable, there are a handful of vetted rules that apply to all observability solutions.

1. Observability tools must be simple to integrate

If integration is too difficult, it's unlikely your project will ever get off the ground. Who has extra cycles to for multi-month integration projects and testing? And, even if there is somehow availability, who wants to take on these sorts of projects?

Successful observability efforts connect with the tools you are already using. They should be able to support polyglot environments (with whatever languages and frameworks you use), integrate with your service mesh or container platform, and connect to Slack and PagerDuty or whatever system your on call team prefers.

2. Observability tools must be easy to use

If the platform is too difficult to learn or use on a daily basis, it won't become part of existing processes and workflows. Developers won't feel

comfortable turning to the tool during moments of high stress, and little improvement will be made in the health and reliability of the system.

3. Observability tools must be built on high-quality telemetry

Upfront sampling, random sampling, and most methods of sampling data simply won't enable observability at scale. High-fidelity data is required to identify outliers or specific issues in a distributed system, as incident resolution may require the analysis of intermittent, infrequent, and rare events.

4. Observability tools must be real-time

Reports, dashboards, and queries need to provide insight into what's going on "right now," so that developers can understand the severity of an issue or the immediate impact of their performance optimizations. When debugging a problem, or fighting an outage, information from ten minutes ago just isn't going to cut it.

5. Observability tools must aggregate and visualize your data

Systems at scale — or even relatively few microservices — produce an enormous amount of data. Far more than can be easily understood by humans without some sort of guidance. For an observability tool to be effective, it needs to make insights obvious. This includes interactive visual summaries for incident resolution and clear dashboards that offer an at-a-glance understanding of any event.

6. Observability tools must provide irrefutable context

What is an investigation without context? Guesswork?

Trial and error? An observability tool should guide its users toward successful incident resolution and system exploration by providing context every step of the way.

- **Temporal context:** How does something look now versus one hour, one day, or one week ago? What did this look like 3 months ago? What did it look like before we deployed?

- **Relative context:** How much has this changed relative to other changes in the system?

- **Relational context:** What is dependent on this, and what is it dependent on? How will changes to this dependency chain affect other services?

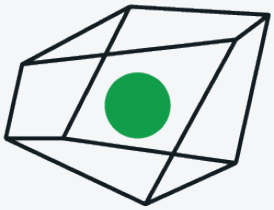
- **Proportional context:** What is the scope of an incident or issue? How many customers, versions, or geographies are affected? Are VIP customers more or less impacted?

7. Observability tools must scale

To enable real-time incident resolution and proactive performance optimization, and observability solution must be able to ingest, process, and analyze your data without latency. Additionally, it can’t be cost-prohibitive to do so, as data goes from terabytes to petabytes and beyond. An effective observability solution must then provide insights across services, which will likely require tracing.

8. Observability tools must provide ROI to the business

This may seem obvious, but it can be all too easy to unintentionally conflate how a tool is perceived by a developer and how it actually does — or does not — drive business value. Ultimately, observability tools must improve the customer experience, increase developer velocity, and ensure a more reliable, resilient, and stable system at scale.



About Lightstep

Lightstep's mission is to deliver confidence at scale for those who develop, operate and rely on today's powerful software applications. Its products leverage distributed tracing technology — initially developed by a Lightstep co-founder at Google — to offer best-of-breed observability to organizations adopting microservices or serverless at scale. Lightstep is backed by Redpoint, Sequoia, Altimeter Capital, Cowboy Ventures and Harrison Metal, and is headquartered in San Francisco, CA. For more information, visit Lightstep.com or follow @LightstepHQ.

Try Lightstep for free

Check out Lightstep's interactive sandbox, and debug an iOS error or resolve a performance regression in less than 10 minutes.