Lightstep

An Engineering Leader's Guide to OpenTelemetry

Daniel "Spoons" Spoonhower

CTO and Co-founder, Lightstep

Making good decisions always starts with having the right data.

Without the right data, you can't be sure what's happening, and you can't understand the outcomes of past decisions. This is especially true for modern software applications, where change is constant: changes to user behavior, changes to infrastructure, and changes to the application itself.

User-facing performance data is some of the most important data you can collect about an application. Users expect applications (whether on a mobile platform or a desktop) to be responsive and correct: if applications are not, they will take their business elsewhere. This usually means measuring things like latency, errors, and critical conversions.

But equally important are the data that describe the *internals* of an application and the infrastructure that it's running on. This is what ultimately shows engineers and developers what they can do to improve user experience or to manage costs. This still means measuring things like latency and error rates, but also being able to map these metrics to individual services, software

components, or even code paths: it means making connections between causes and their effects.

All of this data about the application – or put another way, all of this *telemetry* – will remain important regardless of which languages and frameworks you use and which tools and vendors you choose to help you understand it. In fact, you will absolutely need to evolve your architecture and tools, and good telemetry is a must-have to help you do this.

At the same time, the teams within engineering organizations are gaining more and more autonomy in how they work, both in choosing technologies and tools and in setting the cadences of their work. But regardless of these choices, telemetry must provide a foundation for making decisions that affect the entire application: just because teams are working more independently, it doesn't mean you don't need a global view of what's happening across the application. Good telemetry enables you to reconstruct and compare transactions as they are experienced by users, no matter what technologies or processes your teams are choosing.

The best way for engineering leaders to drive teams toward consistent and scalable choices – ones that are good for the organization as a whole – is to provide a "paved road," a set of frameworks and tools that *make it easy to do the right thing*, while still allowing for some flexibility. However, allowing teams to choose alternatives without setting some standards for

interoperability (particularly in the context of telemetry) can lead to high communication overhead and far-from-optimal outcomes.

OpenTelemetry provides both components that can serve as part of that paved road as well as those that define standards to make sure that divergent choices don't lead to fragmented decision making processes. In this white paper, we'll cover the importance of telemetry in applications, and how that's been amplified with recent technological and organizational changes. We'll discuss why an open, and vendor-neutral approach is critical, and finally, we'll make the case for why OpenTelemetry is the best choice for organizations adopting modern development technologies, tools, and practices.

Why should you care about *telemetry*?

Telemetry is the set data that you use to monitor your applications. It's an umbrella term that covers everything from logs to time series metrics to more recently defined forms of performance data, including distributed traces. Telemetry has been around since the beginning of internet services and, in a broader context, even predates software and goes back to the beginning of electronic *instrumentation*. In fact, in software today, adding instrumentation to an application – whether code changes or just configuration – is how telemetry is generated.

In software, three of the most common types of telemetry are metrics, logs, and distributed traces.

While metrics and logs have been around for a long time, traces have only recently become an important part of the picture as organizations adopted microservices and other loosely coupled architectures.

Metrics are statistics about performance, usually stored and rendered as time series. For example, you might track median request latency for your application over time. Metrics are great at showing you when behavior has changed. Metrics also scale nicely, since storing a metric like median latency requires the same amount of space whether you are processing 10 requests per second or one million.

Logs are records of individual events and might be as simple as a timestamp and a string of text. Ideally, logs have more structure that enables more sophisticated analysis and more efficient searches.

Distributed traces are the newest type of telemetry to be used as part of monitoring software applications. In a sense, traces are just a special kind of structured log: a trace consists of a series of events about a user request as it's handled by different services. Put another way, traces are logs with a built-in index to help you find every event within a distributed transaction and understand how timing and other information explain performance variation.

Because of this history, however, different types of telemetry are often associated with specific tools for collecting, storing, and analyzing them. For example, you'll often hear about a metrics *tool* – a tool that integrates with an application to extract the raw performance data, transmits that data over the network, stores it in a time series database, and then analyzes and visualizes it for users – or a logging *tool* – a tool that collects, stores, indexes, and analyzes logs.

However, decoupling the *generation* of telemetry from the tools used to *analyze* it is important for two reasons. First, different teams may need different sorts of analyses and may choose different tools to do so, especially if your organization is adopting DevOps practices that give teams more flexibility in how they work (often in exchange for taking on more responsibility for managing production services). For example, teams adopting blue-green, canary, or other incremental deployment strategies will need more fine-grained analysis of application metrics than those that are not. Teams using feature-flagging and other experiment frameworks may need to use telemetry both to drive and understand those experiments.

...it's important to decouple telemetry from the tools used to analyze it because siloing different types of telemetry limits the analyses that those tools can perform.

Second and more importantly, it's important to decouple telemetry from the tools used to analyze it because *siloing different types of telemetry limits the analyses that those tools can perform*. Ultimately, telemetry only provides value when it's used as part of monitoring the health of a deployment, improving performance, responding to an incident, or another use case. But if different types of telemetry are siphoned off into different tools – each of which tells just part of the story–engineers will lose the thread. At best it will take longer to resolve incidents as engineers jump from one tool to another, and at worst it will mean that issues are never fully understood and resolved (and will certainly recur at some point in the future).

Telemetry is related to – but different from – another important concept in managing production software systems: *observability*. While the term has a long history in control theory, in the context of software, observability means the ability to *connect cause and effect* in a distributed system. Observability tools help you understand what internal changes have led to a change in user-visible performance or a change in resource consumption.

Observability is often equated with telemetry, for example in a statement like "If you have metrics, logs, and traces, then you have Observability." Observability, however, is really about the process of *deriving value* from telemetry. So while telemetry (possibly in the form of metrics, logs, and traces) is a necessary part of observability, it's not sufficient: you also need tools that will analyze and visualize that telemetry so that you and your organization can take advantage of it.

OpenTelemetry provides a solution to this problem by defining the interface between the application and the tools used to monitor and observe it. Moreover, OpenTelemetry provides APIs and SDKs for logs, metrics, *and* traces. This means that it provides a consistent way to talk about all three types of telemetry.

OpenTelemetry supports the most popular languages – including Java, C#, Go, JavaScript, Python, PHP, Rust, and C++–with more on the way. It describes both semantics and

the syntax, so for example, an HTTP is always identifiable as an HTTP request, regardless of the underlying technology. This means that no matter what languages your teams are using to develop services, OpenTelemetry can serve as a *lingua franca* for understanding performance. OpenTelemetry also defines standards for how telemetry is sent over the network and how context is propagated within an application.

Finally, OpenTelemetry also offers a number of other components, including the OpenTelemetry exporters and collector, that make it easy to get up and running quickly: OpenTelemetry is both the interface *and* an implementation. It provides a means of generating telemetry and a pipeline for routing and processing it as well.

This all means that, while you might choose to give them flexibility in how they implement their services, it's fair to demand that those services generate telemetry in a way that's compatible with OpenTelemetry. By doing so, you're ensuring that you'll be able to get a consistent, global view of the application, and that telemetry won't be siloed according to legacy approaches to collecting and analyzing it.

Why should you choose *vendor-neutral* telemetry?

As DevOps teams take on more and more responsibility for production systems, they need to use – and often learn – new tools to help them manage those systems effectively. Those tools need to be implemented and integrated with existing workflows, and teams need to be trained to use them. And one of the biggest costs – at least in terms of developer effort– associated with new monitoring and observability tools is instrumenting the application to generate telemetry and building a pipeline to ship that data for analysis.

Perhaps you're thinking that the next monitoring or observability tool that your organization adopts will be the last one you'll ever need, and it's a reasonable investment to make that (one last) transition. (Or perhaps you're thinking that this is the last such transition that *you'll* manage in your organization?) Unfortunately, this is not the case. New datastores (and new storage vendors) will mean more efficient ways of storing telemetry. New developments in machine learning will mean that new analyses will offer more value to your teams. (And on the good side, these innovations will offer new value for your organization.) Decoupling telemetry from vendors will mean that any investments your teams make in telemetry will have a much longer lifetime and will help you unlock new value more easily.

When adopting a new tool that analyzes telemetry, there are typically two types of engineering changes required. The first involves instrumentation: changes to the application itself, either in terms of its source code or its configuration. These must typically be undertaken by engineers or developers responsible for that source code, though not always in the case of configuration change. To get the most value from this telemetry, however, it must be deeply integrated with the application at all levels, from infrastructure, to the software lifecycle, to the business logic itself. This means finding a way to express key attributes from each of those layers in a way that can be captured as part of the telemetry.

The second set of changes required when adopting a new tool that analyzes telemetry is setting up the data pipeline, network configuration, and other infrastructure necessary to convey that telemetry from the application to the systems that will store and analyze it. While this is typically only required when moving from one tool to another (or upon major service upgrades), it is still an investment that your infrastructure or operations team will need to undertake.

Integrating your application with OpenTelemetry is ultimately about future-proofing your organization to changes in how telemetry is stored and analyzed. The way that your teams monitor and get insights into application performance *will* change over the next year and beyond. OpenTelemetry provides the abstraction and tools that gives you and your teams the flexibility to adopt new tools when – and how – it's right for them.

Why should OpenTelemetry be *your* standard for telemetry?

We've outlined above how OpenTelemetry is a great way to combine different types of telemetry from across your application and to do so in a way that protects your organization from future vendor changes. But why not set your team out to build a new abstraction for extracting telemetry that is customized to your application and the needs of your organization? After all, your platform team will need to invest in *some* abstraction, why not invest in one that precisely fits your needs? The answer is twofold.

First, the APIs, wire formats, and other standards set by OpenTelemetry have been carefully considered and designed by a panel of industry experts. Even if your team could do incrementally better, you should really ask if it's worth their time to be doing so.

Second, by virtue of its permissive licenses, OpenTelemetry is compatible with other open source projects. It's likely that your application already includes many open source components (either in the form of frameworks that are compiled into proprietary services or as standalone services). If your organization defines its own APIs and formats for telemetry,

these open source components will not be compatible. In fact, those components might already be compatible with OpenTelemetry; if they are not, that is an opportunity for your team to contribute back to the open source projects that you are using.

Your platform team will absolutely need to invest their time in building a paved road for teams to follow. But instead of investing in how telemetry is gathered from the application, have them instead focus on how observability is integrated into other tools. For example, have them consider how your source code repository or continuous integration and continuous delivery (CI/CD) could leverage telemetry to support or automate decision making and other processes.

Conclusion

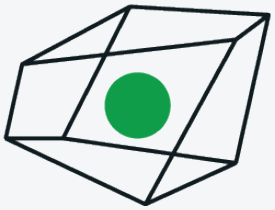
When working with new organizations, we at Lightstep strive to make sure that those organizations are making the best choices for their needs, whether or not that involves Lightstep's solution.

However, we *always* recommend adopting OpenTelemetry as part of projects to modernize how applications are monitored and observed. OpenTelemetry is a low risk way to ensure that your organization can get the most value from observability tools that are available today as well as those that will become available in the future.

Understanding the performance of the application as a whole is absolutely necessary – that's how your users see it. And understanding performance effectively means combining different types of telemetry (metrics, logs, and traces) to derive insights as quickly as possible. At the same time, as your organization continues to adopt DevOps practices that give more autonomy to individual teams, setting standards will become more and more important to avoiding miscommunication and redundant work.

OpenTelemetry provides a cost-effective and scalable way for your platform and infrastructure teams to support the rest of the organization while still enforcing those standards. By using an

expert-built abstraction layer that's compatible with other open source components, your platform team will be able instead to focus on customizations with a higher return-on-investment for your organization, your application, and – ultimately – your users.



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.

Get Lightstep for Free

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