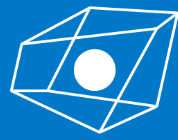
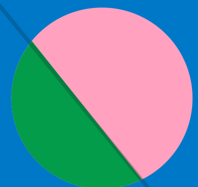


Lightstep

The complete guide to distributed tracing

Daniel “Spoons” Spoonhower

Lightstep CTO and Co-Founder



About Daniel “Spoons” Spoonhower

Lightstep CTO and Co-Founder Spoons (or, more formally, Daniel Spoonhower) has published papers on the performance of parallel programs, garbage collection, and real-time programming. He has a PhD in programming languages but still hasn’t found one he loves.

Follow on Twitter
@save_spoons

Connect on LinkedIn
/spoons

DevOps and distributed architectures

For DevOps to be successful, especially in larger organizations with mission-critical demands, it's imperative to enable loosely coupled work across teams for fast, independent problem solving.

Achieving this requires organizations to build strong abstractions between not only their teams but also their services. On the human level, this means developing common tools, strategies, and best practices for independent, inter-team communication. At the software level, these abstractions take the form of orchestration layers, common build and deployment tools, and shared monitoring services.

But where do these abstractions leave your teams when it comes to observability? How do you understand what your users are seeing? Is that something that can be solved with metrics? What about not just in your software, but in your organization? How do you communicate across teams? How do you find the right team to communicate with during an incident?

These questions and others will be answered as we cover how to establish true observability with distributed tracing:

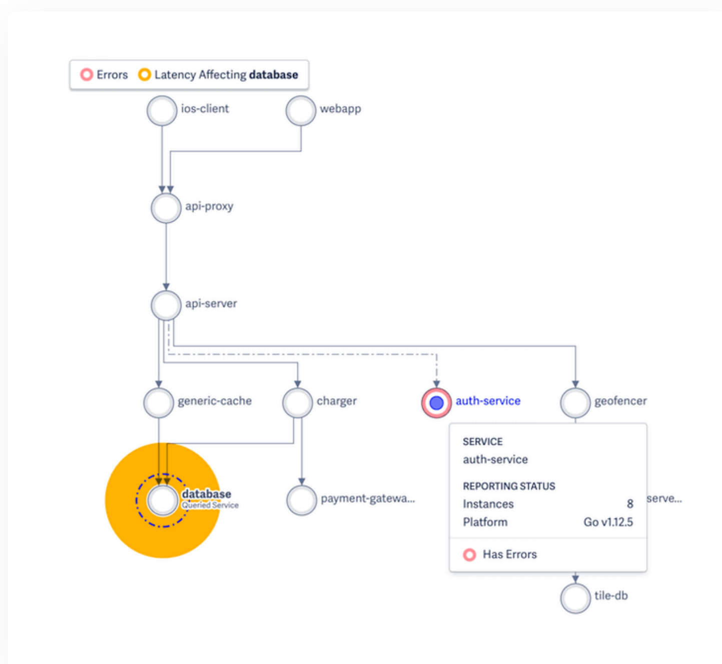
1. How to decide when to take action by focusing on symptoms that directly impact users
2. How to connect cause and effect, whether that means performing root cause analysis during an incident, or proactively improving efficiency and customer experience with performance optimization work

A review of distributed tracing

Let's establish some common language before diving into how to best leverage distributed tracing in your software.

Distrubuted tracing is a diagnostic technique that reveals how a set of services coordinate to handle individual user requests. Tracing is a critical part of observability, as it provides context for other telemetry: for example, helping to define which metrics would be most valuable in a given situation.

Distrubuted tracing is a diagnostic technique that reveals how a set of services coordinate to handle individual user requests.



A single trace shows the activity for an individual transaction or request as it propagates through an application, all the way from the browser or mobile device down to the database and back. In aggregate, a collection of traces can show which backend service or database is having the biggest impact on performance as it affects your users' experiences.

Software architectures built on microservices and serverless offer advantages to application development, but at the same time lead to reduced visibility.

In distributed systems, teams can manage, monitor, and operate their individual services more easily, but it can be more difficult to keep track of global system behavior. During an incident, a customer may report an issue with a transaction that is distributed across several microservices, serverless functions, and teams. Without tracing, it becomes nearly impossible to separate the service that is responsible for the issue from those that are merely affected by it.

With tracing, the focus is on the bigger picture, providing end-to-end visibility, and revealing service dependencies — showing how the services depend on and respond to each other. By being able to visualize transactions in their entirety, you can compare anomalous traces against performant ones to see the differences in behavior, structure, and timing. This information enables you to better identify the culprit and shift directly to addressing a bad change or performance bottlenecks.

Anatomy of a trace

In distributed tracing, a single trace contains a series of tagged time intervals called **spans.** A span can be thought of as a single unit of work. Spans have a start and end time, and optionally may include other metadata like logs or tags that can help classify “what happened.” Spans have relationships between one another, including parent-child relationships, which are used to show the specific path a particular transaction takes through the numerous services or components that make up the application.

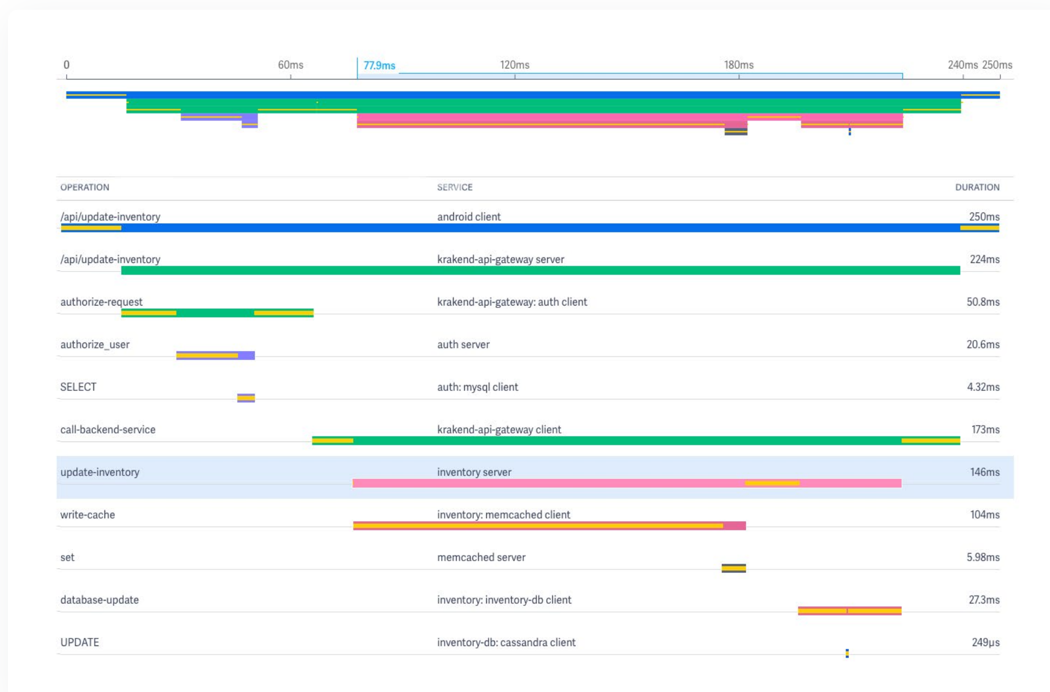
- **Trace**: represents an end-to-end request; made up of single or multiple spans
- **Span:** represents work done by a single-service with time intervals and associated metadata; the building blocks of a trace
- **Tag:** metadata to help contextualize a span

The point of traces is to provide a request-centric view. So, while the architectures that we are choosing enable teams and services to work independently, distributed tracing provides a central resource that enables all teams to understand issues from the user’s perspective.

Understanding trace data

In summary, a single trace represents an end-to-end request and may consist of one or thousands of spans. A span is a timed event with some metadata attached, representing the work done by a particular service as part of this request.

With the basics covered, let's take a closer look at a trace. Each of the lines in the image above represents a span. Time is moving from left to right, so the "start" is on the left and the "finish" is on the right. What other information is provided? We know the duration of the span, or how long it took to "do the work," the name of the service doing the work, and the "operation," which could be the endpoint or some other description of the work that is being done. The yellow line highlights the combination of work that determined how long it took the overall action to complete, usually called the "critical path."



For a more detailed look of the mechanics of tracing, check out this technical report written by Lightstep CEO and Co-Founder Benjamin Sigelman about Dapper, a large-scale distributed tracing infrastructure he developed while at Google.

Let users drive decision making

A few questions for service owners, engineers, and operators: What do your users expect from your application? How are they going to measure your success? And if you don't hit those targets, or if you don't meet those user expectations, what will they do? Are you prepared to handle these failures? These questions are vital for addressing service performance, and their answers can be formalized through **service levels**.

Service levels provide an important context for what to prioritize when responding to shifting performance demands—and distributed tracing can help you manage your service levels.

Service levels

- **Service Level Indicator (SLI):** measurable data such as latency, uptime, and error rate
- **Service Level Objective (SLO):** defines the target for SLIs. For example, p99 latency < 1s; 99.9% uptime; <1% errors
- **Service Level Agreement:** financial or contractual consequences for failing to meet SLOs. Refunds, cancellations, etc

From an engineering point of view, **service levels are important because they reflect customer expectations.** What metrics do your customers care about? What kinds of failures will stand in the way of your customers' success?

Choosing the correct scope for your service levels

What's the correct level of granularity for your SLI? Is it at the operation level? At the service level? It might be easy to think, "Well, I own Service X, so I'm going to set an SLI for that service to track performance. Latency is important for my users, so I will track the latency of Service X." It is important to pause here and think about the different kinds of things that your service does.

Let's look at a simple example: you have a single service that handles two endpoints, one that reads and one that writes. The read endpoint is likely going to be a lot faster than the write endpoint. You can set the Service Level Indicator (SLI) wide enough to cover both, but you will lose insight into the faster read operation. Alternatively, you can set it tight enough to cover the read, but you will be constantly missing your Service Level Objective (SLO) because it won't be taking the write operation into consideration. So for a service like this, it might be best two set two independent SLIs: one for each operation.

Measure the correct type of performance

Once scope is settled, the next step is choosing the right types of performance to measure:

- **High Percentile Latency**: outlier latency issues can 'bubble' up the stack and compromise performance
- **Error Rate**: rate of errors in the system can offer real-time alerts that there is a problem
- **Throughput**: setting a service level around throughput is tricky, because it can be independent of user behavior, but is ultimately still an important measure of system health

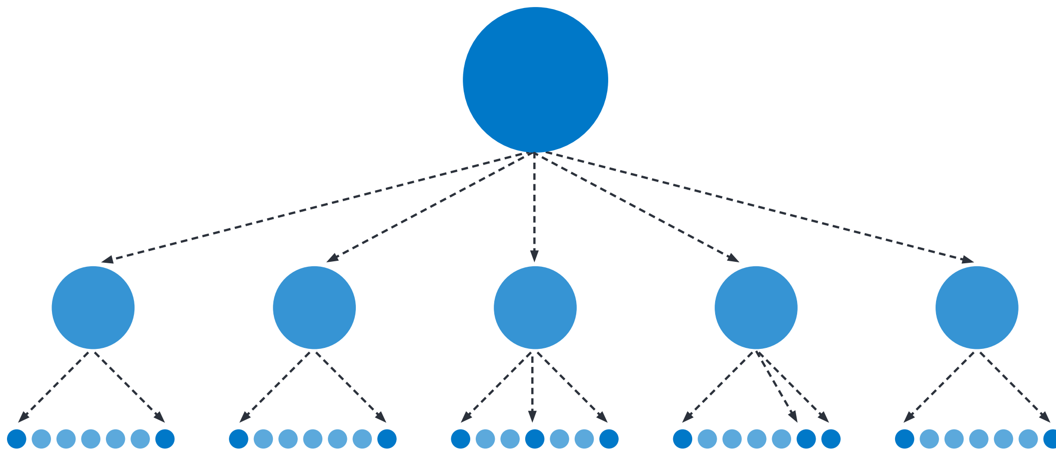
- **Saturation**: what percentage of your resources are being utilized to handle traffic? This is especially important for understanding if database or shard resources need to be scaled up or altered

When choosing your SLIs, **focus on what is critical to your users**. Imagine a software company handling “low frequency” trading, essentially replacing a pen and paper process for fulfilling trades. Because this company’s services are replacing such a slow, manual process, the service might be down for 10 to 20 minutes without much complaint. Correctness, on the other hand — making the right trades at the right time — is paramount to this company’s users. It would then seem wise to focus resources on achieving 100% correctness at the expense of uptime, if necessary.

Understanding latency percentiles in distributed systems

Latency percentiles are especially important in distributed systems.

Suppose the graphic below is a simple representation of a search system that spans out across many shards.



Let's say the services at the very bottom of the stack have an average latency of one millisecond, but the 99th percentile latency is one second.

To be clear, a 99th percentile latency of one second means that one percent of the traces are one thousand times slower than the average request (1 ms).

Now if an end user request—that is, something coming from the top of this system—hits only one of those bottom services, one percent of those end user requests will also take a second or more. But there are some subtleties to take note of here: as the complexity of handling those requests increases, say, by increasing the number of service instances at the bottom of the stack that have to participate

in that request, this 99th percentile latency is going to have a bigger and bigger impact.

If a request requires a hundred of these bottom-of-the-stack services to participate in answering one of these search queries, then 63% of end user requests are going to take more than a second. So, even though only one percent of the requests at the very bottom of the stack are experiencing this one second latency, more than half of the requests that the users initiate would take more than a second.

The takeaway? As our systems increase in complexity, “rare behaviors” like 99th percentile latency can get magnified through that complexity. Accordingly, one of the really important reasons to be measuring 99th percentile metrics throughout the entire stack is to understand how those latencies are going to be magnified as as they travel back up the stack.

For a more in-depth overview on system complexity, check out Ben Sigelman’s overview and webinar on Deep Systems.

Connecting cause and effect with distributed tracing

The same way a doctor first looks for inflammation, reports of pain, and high body temperature in any patient, it is critical to understand the symptoms of your software’s health. Is your system experiencing high latency, spikes in saturation, or low throughput? These symptoms can be easily observed, and are usually closely related to SLOs, making their resolution a high priority.

Once a symptom has been observed, distributed tracing can help identify and validate hypotheses about what has caused this change.

It is important to use symptoms (and other measurements related to SLOs) as drivers for this process, because there are thousands—or even millions—of signals that *could* be related to the problem, and (worse) this set of signals is constantly changing. While there might be an overloaded host somewhere in your application (in fact, there probably is!), it is important to ask yourself the bigger questions: Am I serving traffic in a way that is actually meeting our users' needs? Is that overloaded host actually impacting performance as observed by our users?

In the next section, we will look at how to start with a symptom and track down a cause. Spoiler alert: it's usually because something changed.

Change drives outages

Service X is down. What happened? As a service owner your responsibility will be to explain variations in performance—especially negative ones. A great place to start is by finding out what, if any, changes have been made to the system prior to the outage. Sometimes it's internal changes, like bugs in a new version, that lead to performance issues. At other times it's external changes—be they changes driven by users, infrastructure, or other services—that cause these issues.

Changes in the service

Perhaps the most common cause of changes to a service's performance are the deployments of that service itself. Distributed tracing can break down performance across different versions, especially when services are deployed incrementally. This, in effect, surfaces the latency, error rate, and throughput for all of your services endpoints, and allows you to understand changes to performance before, during, and after your deploy hits production.

This is made possible by tagging each span with the version of the service that was running at the time the operation was serviced. (Traces are all about context: Spans can also be tagged by geography, operating system, canary, user_id, etc.)

Changes in user demands

Changes to service performance can also be driven by external factors. Your users will find new ways to leverage existing features or will respond to events in the real world that will change the way they use your application. For example, users may leverage a batch API to change many resources simultaneously or may find ways of constructing complex queries that are much more expensive than you anticipated. A successful ad campaign can also lead to a sudden deluge of new users who may behave differently than your more tenured users.

Being able to distinguish these examples requires both adequate tagging and sufficient internal structure to the trace. Tags should capture important parts of the request (for example, how many resources are being modified or how long the query is) as well as important features of the user (for example, when they signed up or what cohort they belong to).

In addition, traces should include spans that correspond to any significant internal computation and any external dependency. For example, one common insight from distributed tracing is to see how changing user behavior causes more database queries to be executed as part of a single request.

Infrastructure and resource competition

All the planning in the world won't lead to perfect resource provisioning and seamless performance. And isolation isn't perfect: threads still run on CPUs, containers still run on hosts, and databases provide shared access. Contention for any of these shared resources can affect a request's performance in ways that have nothing to do with the request itself.

As above, it's critical that spans and traces are tagged in a way that identifies these resources: every span should have tags that indicate the infrastructure it's running on (datacenter, network, availability zone, host or instance, container) and any other resources it depends on (databases, shared disks). For spans representing remote procedure calls, tags describing the infrastructure of your service's peers (for example, the remote host) are also critical.

With these tags in place, aggregate trace analysis can determine when and where slower performance correlates with the use of one or more of these resources. This, in turn, lets you shift from debugging your own code to provisioning new infrastructure or determining which team is abusing the infrastructure that's currently available.

Upstream changes

The last type of change we will cover are upstream changes. These are changes to the services that your service depends on. Having visibility into your service's dependencies' behavior is critical in understanding how they are affecting your service's performance. Remember, your service's dependencies are — just based on sheer numbers — probably deploying a lot more frequently than you are. And even with the best intentions around testing, they are probably

not testing performance for your specific use case. Simply by tagging egress operations (spans emitted from your service that describe the work done by others), you can get a clearer picture when upstream performance changes. (And even better if those services are also emitting spans tags with version numbers.)

Simple guidelines for tagging

Understanding these changes will help you plan a more robust instrumentation. What is the correct granularity for spans? And what are the correct tags to use within those spans to enable automated analysis?

We can establish some simple guidelines about what information to track, depending on the general type of operation:

- **All Operations**: software version, infrastructure info
- **Ingress and Egress Operations:** peer info, key request parameters, response code
- **Large Components/Libraries**: domain-specific tags

Include spans for any operation that might independently fail (remote procedure calls should immediately come to mind) or any operation with significant performance variation (think database query) or that might occur a variable number of times within a single request.

You might think, “can’t this be accomplished with the right dashboard tooling?” Remember that your goal as a service owner is to explain variation in performance, and to do so, you must not just enumerate all of the metrics, but quickly identify the metrics that matter—with an emphasis on *quickly*. This is where tracing shines, because it draws a clear line between your SLIs and the metrics which explain variation in these SLIs—whether those metrics are part of your service or a service that belongs to another team.

Proactive solutions with distributed tracing

So far we have focused on using distributed tracing to efficiently *react* to problems. But this is only half of distributed tracing's potential. How can your team use distributed tracing to be proactive?

The first step is going to be to establish ground truths for your production environments. What are the average demands on your system? With the insights of distributed tracing, you can get the big picture of your service's day-to-day performance expectations, allowing you to move on to the second step: improving the aspects of performance that will most directly improve the user's experience (thereby making your service better!).

- **Step One**: establish ground truths for production
- **Step Two:** make it better!

The following are examples of proactive efforts with distributed tracing: planning optimizations and evaluating SaaS performance.

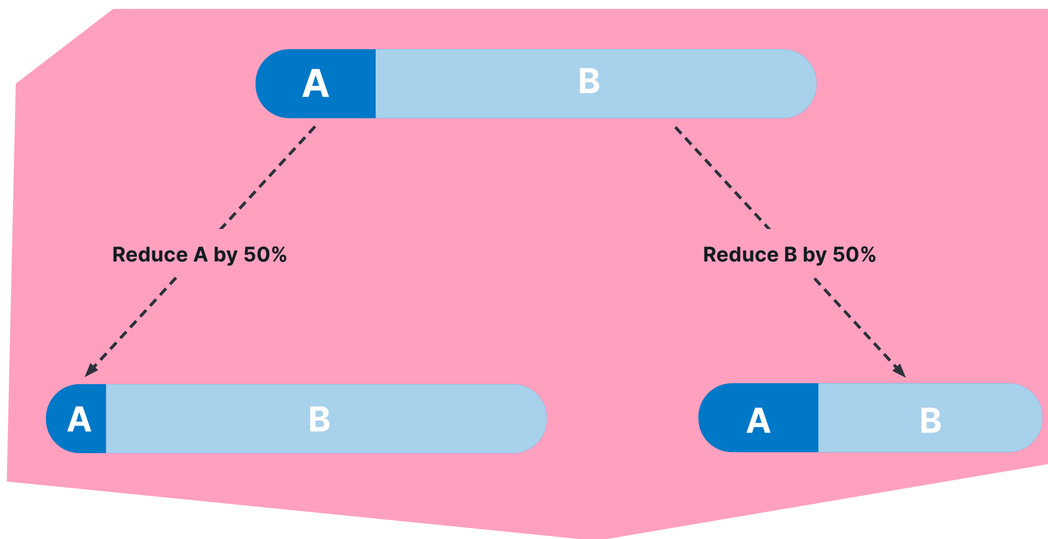
Planning optimizations: how do you know where to begin?

Your team has been tasked with improving the performance of one of your services — where do you begin? Before you settle on an optimization path, it is important to get the big-picture data of how your service is working. Remember, establish ground truth, then make it better!

Answering these questions will set your team up for meaningful performance improvements:

- **What** needs to be optimized? Settle on a specific and meaningful SLI, like p99 latency.
- **Where** do these optimizations need to occur? Use distributed tracing to find the biggest contributors to the aggregate critical path.

In the image below, we have a trace with two operations: A and B. Operation A takes about 15% of the time of the overall trace, while B consumes the rest.



With this operation in mind, let's consider Amdahl's Law, which describes the limits of performance improvements available to a *whole* task by improving performance for *part* of the task. Applying Amdahl's Law appropriately helps ensure that optimization efforts are, well, optimized.

What Amdahl's Law tells us here is that focusing on the performance of operation A is never going to improve overall performance more than 15%, even if performance were to be fully optimized. If your real goal is improving the performance of the trace as a whole, you need to figure out how to optimize operation B.

In short: Don't waste time or money on uninformed optimizations.

Evaluating managed services

Managed services provide a lot of flexibility to engineering teams, enabling time-intensive services such as storage, analysis, and load balancing to be offloaded; and, for services with a lot of exposure, such as fraud and abuse detection, authentication, and payments, managed services can offer meaningful security. Regardless of how it fits into the application, from an observability point of view, a managed service is ultimately just another service, and the same design and testing considerations need to be made.

While nearly all managed services publish SLAs (yours do, right?), no vendor is going to care as much as you do about how they can affect your service's performance. Measure them like any other service in your application. Use best practices for things like provisioning. Even if you can't instrument the managed service itself, instrumenting SDKs, and monitoring them just like you would any of your other upstream dependencies, goes a long way toward better visibility.

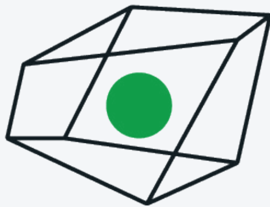
Summary

While there are no silver bullets in the software performance, distributed tracing provides unrivaled visibility and analysis in systems that can have dozens, hundreds, or thousands of services working together. With features like tags that enable automated analysis, traces provide request-centric views into system health and performance pivotal for a DevOps organization.

When deciding what performance metrics to focus on, prioritize the needs and expectations of your software's users. Service levels help formalize these expectations, and provide a focus for investment and engineering efforts when triaging an issue or making optimization decisions.

As issues do arise, use distributed tracing to isolate and highlight changes that impact your SLIs. These changes can be internal, like new deployments, but are often external, including your service's dependencies or user behavior.

Empower your DevOps organizations with distributed tracing for fast root-cause analysis and performance optimization in deep systems.



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.

©2020 Lightstep, Inc.