Lightstep

Best practices for root cause analysis

Executive Summary

Systems fail. These failures and outages kick off an all-too-common chain of events: upset customers, missed SLOs, lost revenue, and stressed out developers.

What sets companies apart is how they learn from outages, to more effectively and efficiently find the root cause of problems.

Root cause analysis is about understanding not just what happened but why it happened. It's about how our assumptions about a system or services are different from reality, so that fixes address the underlying cause instead of simply rolling back the latest deployment. Effective root cause analysis helps keep customers happy, prevent lost revenue, enable continuous development velocity, improve organizational efficiency and over time, might build a more resilient tech stack.

Best practices for root cause analysis

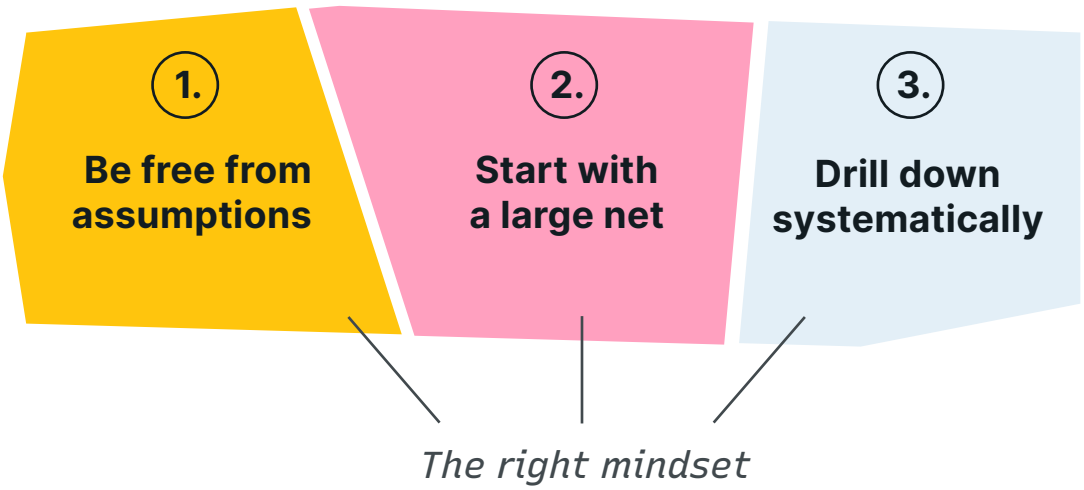
Once you're at the point of investigating the root cause of a problem, it's likely because someone was complaining or something internally set off an alert. You might have had customers who are having a poor experience with an application, either because it's slow, some features weren't working or they can't access it at all. During that outage, the team's focus is on getting the application back up. That might mean first rolling back to a previous version or finding another work-around. That's great: you should absolutely be working to first mitigate the problem.

Once the problem has been mitigated, however, it's time to really get to work: again, root cause investigations aren't about fixing outages. They're about understanding why the outage happened in the first place so that teams can prevent this from occurring in the future.

The right mindset

It's important to start root cause analysis with the right mindset. This means:

1. **Free from assumptions.** Everyone involved should stay open-minded to what the root cause might be, and use evidence to prove or disprove hypotheses.
2. **Start with as large a net as possible.** You'll want to examine or have a tool that examines as many possible factors as possible. This means looking at different types of changes as well as a wide time frame — sometimes the root cause of a problem was a change that happened significantly before the incident occurred.
3. **Drill down systematically.** Your analysis should drill down with increasing granularity as you uncover more evidence. The more granular the data, the more likely you'll be able to identify and remedy the root cause. Likewise, you'll want to be wary of overly sampled datasets and performance monitoring and observability tools that do not capture 100% of your event data.



Avoiding tribal knowledge

Probably the most common mistake when it comes to root cause analysis is overreliance on tribal knowledge. This often happens when organizations don't have robust observability tools in place that would allow them to take a more methodical approach to investigations, starting from the big picture, looking for clues and zeroing in with increasing granularity. Here's why tribal knowledge is such a big problem.

It can't scale.

Relying on individuals to 'know' what happened might work if there's one engineer and one or two services. But as complexity increases, it becomes simply impossible for any person to keep an accurate mental picture of how the system behaves in the wild.

It's inaccurate.

Relying on what a single engineer or team 'thinks' they know about the application can cause teams to ignore any evidence that contradicts the theory. This is especially a problem in edge use cases: If the root cause of a problem is X nine times out of ten, engineers will always assume that X is the problem. In some cases, this mindset will make it either impossible or exceptionally time consuming to find the real root cause. When relying on tribal knowledge to find a root cause, teams are prevented from keeping

an open mind about the wide variety of root causes that could be the underlying root cause — from server issues to an edge-case dependency to a change in usage patterns.

It creates business risk.

Relying on tribal knowledge holds organizations hostage to their most senior engineers — the 'holders' of the tribal knowledge. If those engineers get a better job offer, retire or burn out, organizations can instantly lose all of their tribal knowledge, and find themselves unable to find root causes.

It creates burn-out.

The engineers who hold the most tribal knowledge are also the organization's most senior, most experienced engineers. If they are constantly being pulled into investigations, they are distracted from working on other projects. Their own development velocity decreases — they spend less time and energy finding novel ways to use software to create value for the organization. It can also be frustrating for those engineers, who would rather be creating value for the company.

But how do organizations get away from tribal knowledge in root cause analysis? By using the right tools to understand context, focusing on actionable information, and preparing proactively.

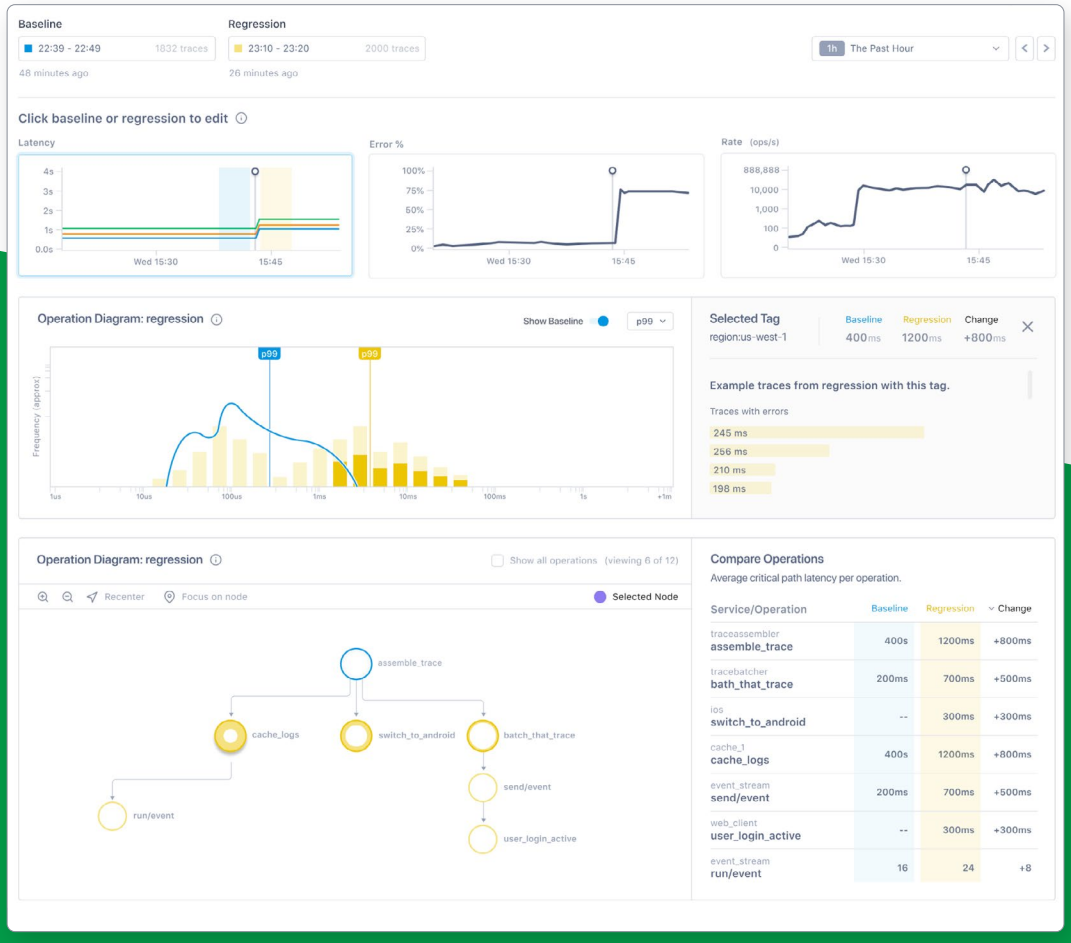
Best practices for effective root cause analysis

Once you are ready to get started, you can follow some of these best practices for efficiently root causing a problem:

- **Understand the context**
- **Focus on what you can actually control**
- **Continuously improve processes**

Understand the context

When looking for the root cause of a problem, it's important to understand not just how individual components of a system work but also how they interact with each other; this means understanding how your service interacts with other services. This allows teams to start with a big picture and follow connections progressively as they look for indications of where a root cause is coming from. Contextual information is critical in identifying how services behave and provide everyone with a better mental map of how they communicate and depend on one another.



Context is critical. Root cause analysis tools need to not only capture and present data, but surface meaningful insights and correlations that help developers understand exactly why a change in performance occurred.

Look for connections between seemingly unrelated events.

The dependencies in modern applications are complicated, and engineers often don't understand them all. This is especially true in large organizations where teams are responsible for one service and have little overlap with the teams responsible for other services. This process can involve:

- Looking for alerts and/or errors that happened in succession
- Looking for outside factors, like changes in customer usage or announcements from your cloud provider
- Using tools to get an accurate dependencies map

Finding connections and putting the problem into the correct context is key to building a more accurate mental map of how the application behaves in the real world.

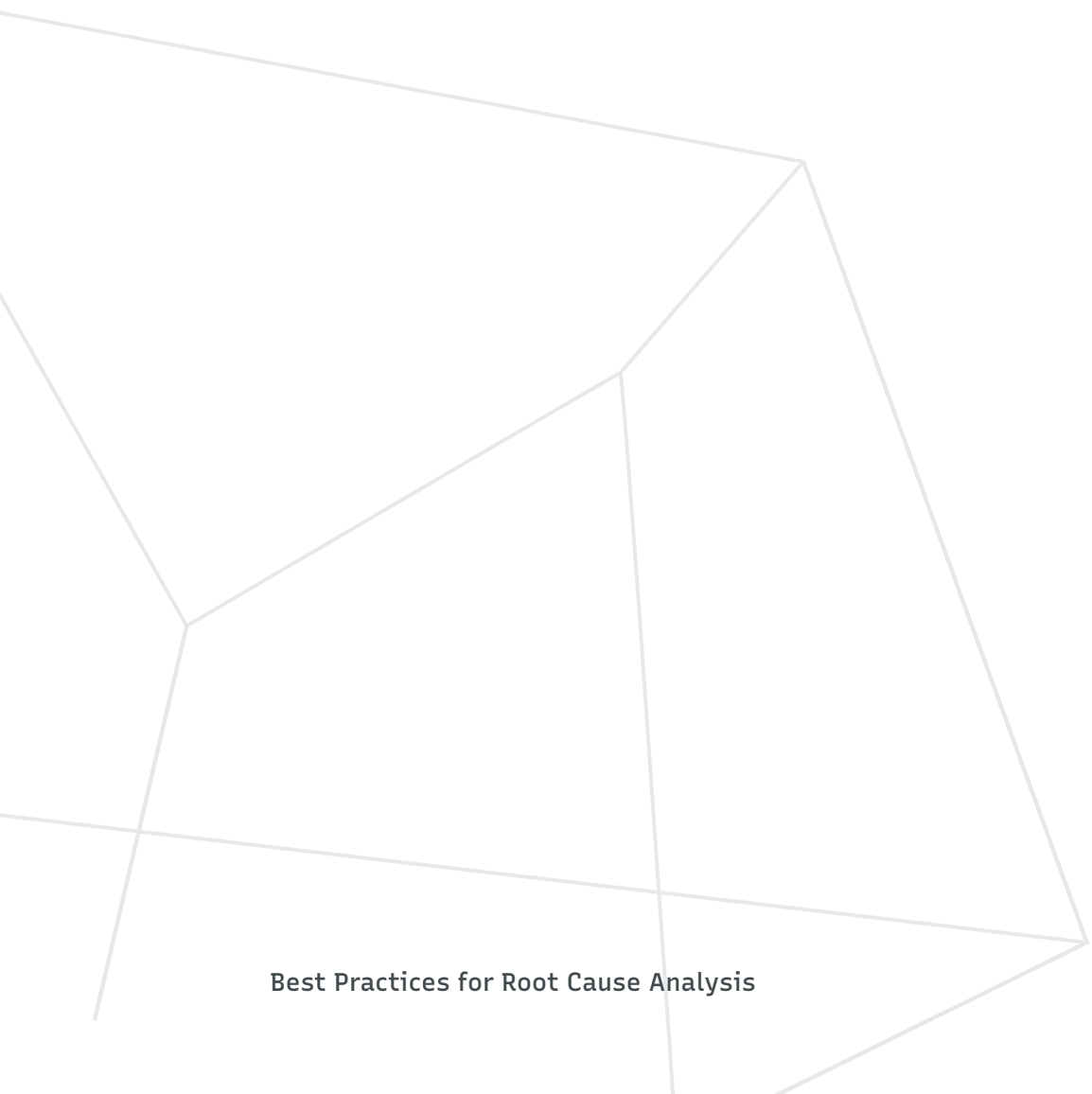
Narrow the search space.

Effective root cause analysis isn't about looking at every data point. In fact, it's quite the opposite. You'll want to use tools that automatically surface the likely root cause(s) and identify non-performant signals that relate to or correlate with a specific incident.

Manually combing through logs or other large data sets simply takes too long, especially when the root cause is often the result of a specific deploy, customer action, or dependency that you may or may not even be aware of.

Track changes back in time.

Once you've identified something that is clearly correlated with the issue, it's important to track it back in time and see if you can find the first time the error occurred — even if the first occurance wasn't noticed by anyone. While you narrow down the time window in which to look for changes, you should also keep an open mind about what the root cause could be. For example, it could turn out that a library was updated right before the error started showing up. Or, it's possible that a customer altered its usage behavior.



Focus on what you can actually control: actionable causes

In a certain number of cases, the root cause of an outage is something completely out of the organization's control — or at least it seems that way. During the investigation however, it is almost always possible to make even seemingly 'unactionable' root causes fixable. For example:

- **The outage was caused by a cloud provider error.** This wasn't technically the team's fault, but there are ways to architect the application so that it's able to automatically fail over to other availability zones and/or another cloud provider if one goes down.
- **The root cause was a change in customer behavior.** It would be a good idea to reach out to the customer to make sure the change in usage patterns is intentional, consider updating your terms of service and/or finding a different way to manage scaling so that similar changes in usage don't cause an outage.

In the scenarios above, organizations may or may not decide to make the fix. It might not make business sense to re-architect the application to avoid two minutes of downtime per month — all depending on how critical the application is and what the consequences of downtime are. Not all root causes are easily reversible, especially if they happened too far in the past. In those

cases, organizations have to see what can be done to mitigate the potential for the same root cause to cause problems again.
It's also important to remember that the 'root cause' of a problem was not necessarily a 'mistake.' The reason root cause analysis is so important is because engineering teams don't always understand how the application actually functions in a real-life, production environment, with real users and under various edge cases. Even if the root cause was a mistake, root cause analysis sessions should be 'blameless.' The goal of root cause analysis isn't to point fingers at the individual or team who caused the mess — it's to clean up and make sure it doesn't happen again. If engineers are worried about being blamed for an incident, they're less likely to be as engaged in the root cause analysis process and less likely to be completely honest.

Clear mistakes are generally easier to fix than root causes that force teams to reconstruct their mental map of how the application components work together.

Control the future with chaos gamedays

The best way to improve an organization's root cause analysis process (as well as the incident response) is to practice. Running Chaos GameDays is a way to proactively understand how your system will respond under a variety of circumstances and to keep everyone's mental map of the system architecture as up-to-date as possible.

The goal of a GameDay is to answer questions like 'what could go wrong in X scenario?' or 'do we know what will happen if X service goes down?' The type of information teams need during and after a GameDay are the same that they'll need to do root cause analysis after a failure.

During the GameDay, teams should ask themselves the following questions:

- Do we have enough information?
- Is the application behaving in the way we expected, given the type of failure we're testing?
- What would the end user experience if this failure were to happen in production?
- How are upstream and downstream dependencies impacted?
- Are the upstream and downstream dependencies behaving in the way we expected?

After the GameDay, it's important to do a post-mortem — including a GameDay root cause analysis. Like a post-mortem following an incident, this should be done within a couple of days after the GameDay, and should include any steps you might take after an incident, including:

- Making changes to your monitoring and observability tools
- Reporting a bug
- Updating the runbook
- Running a complete root cause analysis, especially if something happened that the team didn't understand.

GameDays can also be a part of the root cause analysis process. After changes have been made, they can help determine if the root cause identified and fixed was in fact the root cause.

- If the GameDay uncovers additional unexplained behavior and/or the attempted fix has no effect, it's back to the drawing board
- If the GameDay verifies that the root cause was correctly identified and has been corrected, it's important to set up some kind of automated testing to ensure that subsequent deployments don't re-introduce the failure.

Running regular GameDays gives teams both more practice with root cause analysis as well as information about what tools and practices they need to implement proactively so that post-incident root cause analysis can be as successful and smooth as possible.

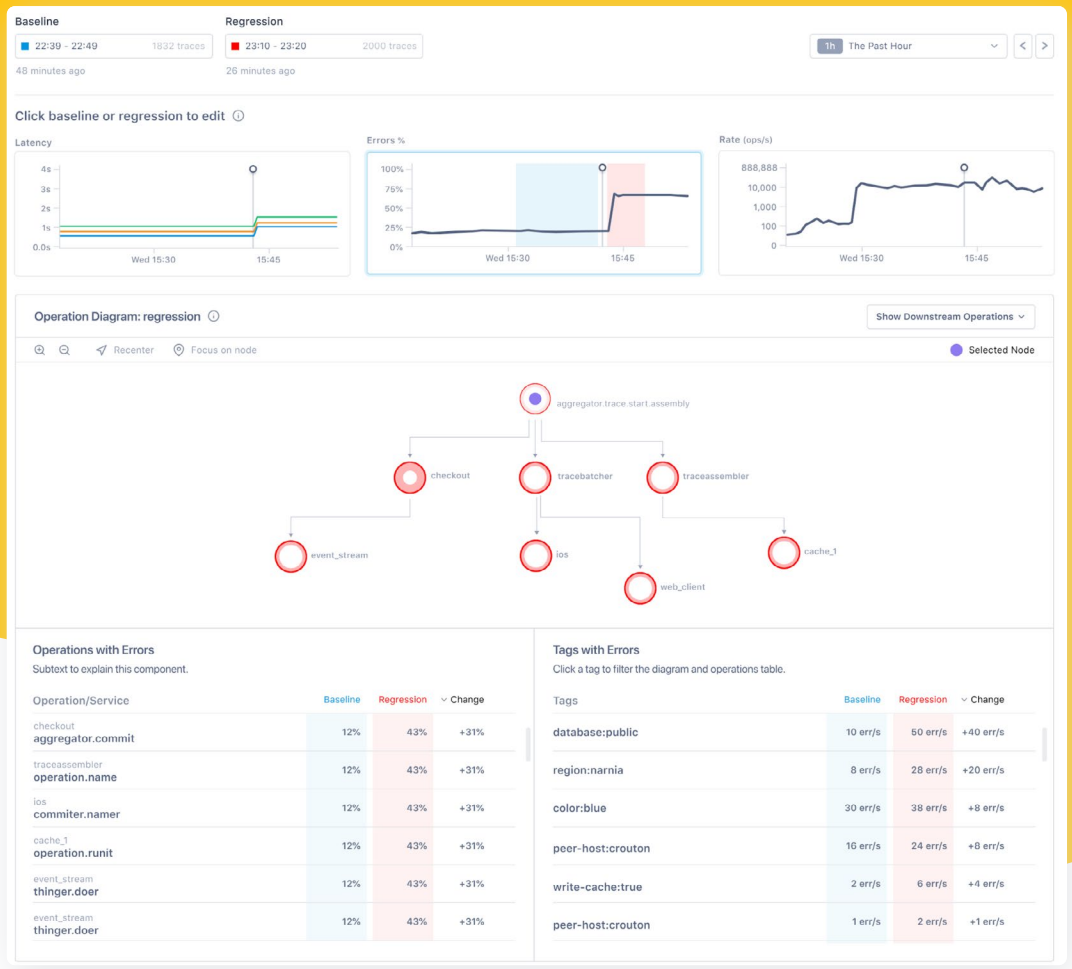
The Results: What Effective Root Cause Analysis Does

Adjust the mental map

Every engineer in an organization makes a series of assumptions about how the application works, what the network of dependencies looks like, and what the most likely causes of an incident will be. This understanding of how each component of the application will behave under various circumstances is based on experience both with other applications and in the engineer's current role.

Even the most senior, experienced engineer will not have an entirely accurate understanding of the application's behavior under all circumstances. Modern applications are complicated and dynamic, so even if an engineer had a series of assumptions that were correct at one point, they will be out-of-date within days, as services are updated and new features are added.

Root cause analysis gives teams a way to update their assumptions about how the application really works. The better the root cause analysis process, the more accurate team members' mental picture of the application will be. This will ultimately make it not only make it easier to fix future incidents, but will also make them less likely to happen — because developers will better understand how services are related and can avoid potential problems at the development stage.



Adjust your mental map. Understanding operation and service relationships is fundamental to root cause analysis in distributed systems. RCA tools should help provide a clear model of these dependencies.

Done right, root cause analysis will uncover the delta between what you 'think' happened and what actually happened. By extension, it uncovers the delta between what 'is' happening at any time — when the application is running as planned, as well as when there is a problem — and what team members 'think' is happening.

Avoid burnout

Fixing the same problem repeatedly is demoralizing. Bugs are a fact of life as a developer, but making the same mistakes is frustrating on a number of levels.

Incidents are disruptive, especially for the developers involved in incident response. Being woken up in the middle of the night is never fun. When it's because of something that has happened before and could have been avoided with careful root cause analysis, the bug can feel like a symptom of organizational dysfunction — the kind that makes engineers want to look for another job or take a sabbatical.

Conclusion

Root cause analysis should be done shortly after an incident, when it's fresh in everyone's mind. Having the right metrics and data visualizations are important, but so is human intelligence and getting as many human perspectives as possible.

Root cause analysis depends on having access to enough information about the system architecture, upstream and downstream dependencies, and runtime behavior, but finding the root cause requires human intelligence, organizational diligence, and the right mindset.

A methodical, data-driven approach to root cause analysis will improve the application's resilience, as each incident makes the application stronger and increases awareness about runtime behavior. It doesn't ensure an end to all incidents. Each incident will likely be more difficult to fix, although the incidents themselves should be increasingly rare.

The bottom line, however, is that every root cause analysis should be a fresh mystery to solve, uncovering a new root cause for a new type of issue every time.

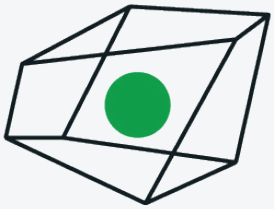
Sidebar: The ‘Tired Engineer’ Root Cause

The goal of root cause analysis is ultimately to get to the bottom of an outage, slowdown or other problem from a technical perspective, so you and the entire organization can better understand how the application really works. Sometimes the technical cause could not have been spotted ahead of time, because it wasn’t really a mistake — it was code or a dependency that would have worked fine if everyone’s assumptions about the application had been accurate.

Other times the root cause was clearly a mistake. When that happens, it’s appropriate to ask how the mistake happened — to look for organizational root causes. Some examples might be:

- There was pressure from the business to ship this application before it had been properly tested
- The engineer was working her third 80-hour week and was tired
- The team was skipping code reviews and continuous integration testing

Thinking about organizational root causes is a valid part of the root cause analysis exercise, but should not obscure the true goal of root cause analysis: To increase everyone’s understanding of how the application works in real life and to ensure that errors are fixed in a way that makes the application more resilient over time.



About Lightstep

Lightstep enables teams to detect and resolve regressions quickly, regardless of system scale or complexity. We integrate seamlessly into daily workflows, whether you are proactively optimizing performance or investigating a root cause so you can quickly get back to building features.

Want to see Lightstep in action?

Want to see Lightstep in action? Try our interactive sandbox, and resolve a performance regression in minutes.