O'REILLY®

Anatomy of an Incident

Google's Approach to Incident Management for Production Services

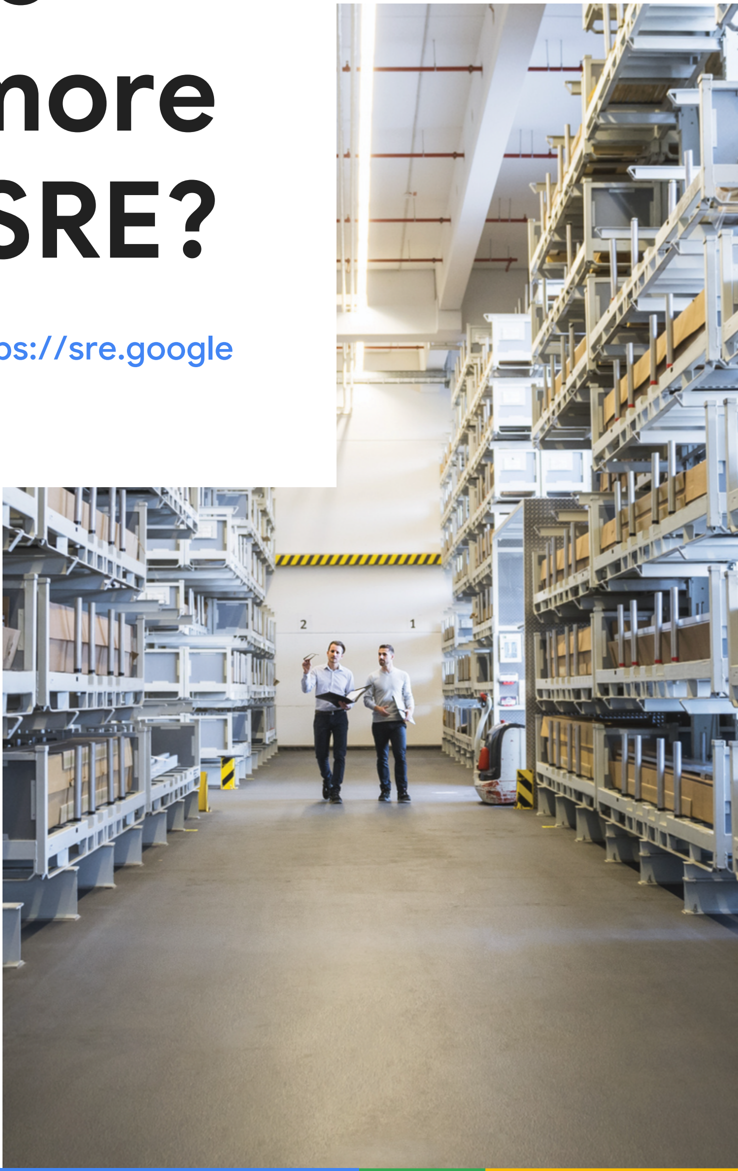
Ayelet Sachto & Adrienne Walcer
with Jessie Yang

REPORT

Google Cloud

Want to know more about SRE?

To learn more, visit https://sre.google



Anatomy of an Incident

Google's Approach to Incident Management for Production Services

Ayelet Sachto and Adrienne Walcer,
with Jessie Yang

Beijing • Boston • Farnham • Sebastopol • Tokyo
O'REILLY®

Anatomy of an Incident
by Ayelet Sachto and Adrienne Walcer, with Jessie Yang

Printed in the United States of America.

Published by O’Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisition Editor: John Devins
Development Editor: Virginia Wilson
Production Editor: Beth Kelly
Copyeditor: Audrey Doyle

Proofreader: Piper Editorial Consulting, LLC
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

January 2022: First Edition

Revision History for the First Edition
2022-01-24: First Release

The views expressed in this work are those of the authors and do not represent the publisher’s views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

This work is part of a collaboration between O’Reilly and Google. See our statement of editorial independence.

978-1-098-11372-8

[LSI]

Table of Contents

CHAPTER 1
Introduction

> Make no mistake—the coming N weeks are going to be personally and professionally stressful, and at times we will race to keep ahead of events as they unfold. But we have been preparing for crises for over a decade, and we're ready. At a time when people around the world need information, communication, and computation more than ever, we will ensure that Google is there to help them.
>
> —Benjamin Treynor Sloss, Vice President, Engineering,
> Google's Site Reliability Engineering Team, March 3, 2020

Failure is an inevitability (kind of depressing, we know). As scientists and engineers, you look at problems on the long scale and design systems to be optimally sustainable, scalable, reliable, and secure. But you're designing systems with only the knowledge you currently have. And when implementing solutions, you do so without having complete knowledge of the future. You can't always anticipate the next zero-day event, viral media trend, weather disaster, config management error, or shift in technology. Therefore, you need to be prepared to respond when these things happen and affect your systems.

One of Google's biggest technical challenges of the decade was brought on by the COVID-19 pandemic. The pandemic created a series of rapidly emerging incidents that we needed to mitigate in order to continue serving our users. We had to aggressively boost service capacity, pivot our workforce to be productive at home, and build new ways to efficiently repair servers despite supply chain constraints. As the quotation from Ben Treynor Sloss details, Google was able to continue bringing services to the world during this paradigm-shifting sequence of incidents because we had prepared for it. For more than a decade, Google has proactively invested in incident management. This preparation is the most important thing an organization can do to improve its incident response capability. Preparation builds resilience. And resilience and the ability to handle failure become a key discipline for measuring technological success on the long scale (think decades). Beyond doing the best engineering that you can, you also need to be continually prepared to handle failure when it happens.

Resiliency is one of the critical pillars in company operations. In that regard, incident management is a mandatory company process. Incidents are expensive, not only in their impact on customers but also in the burden they place on human operators. Incidents are stressful, and they usually demand human intervention. Effective incident management, therefore, prioritizes preventive and proactive work over reactive work.

We know that managing incidents comes with a lot of stress, and finding and training responders is hard; we also know that some accidents are unavoidable and failures happen. Instead of asking "What do you do *if* an incident happens?" we want to address the question "What do you do *when* the incident happens?" Reducing the ambiguity in this way not only reduces human toil and responders' stress, it also improves resolution time and reduces the impact on your users.

We wrote this report to be a guide on the practice of technical incident response. We start by building some common language to discuss incidents, and then get into how you encourage engineers, engineering leaders, and executives to think about incident management within the organization. We aim to cover everything from preparing for incidents, responding to incidents, and recovering from incidents to some of that secret glue that maintains a healthy organization which can scalably fight fires. Let's get started.

What Is an Incident?

Incident is a loaded term. Its meaning can differ depending on the group using it. In ITIL, for example, an incident is any unplanned interruption, such as a ticket, a bug, or an alert. No matter how the word is used, it's important that you align on a specific definition to reduce silos and ensure that everyone is speaking the same language.[1]

At Google, incidents are issues that:

- Are escalated (because they're too big to handle alone)
- Require an *immediate* response
- Require an organized response

Sometimes an incident can be caused by an *outage*, which is a period of service unavailability. Outages can be planned; for example, during a service maintenance window in which your system is intentionally unavailable in order to implement updates. If an outage is planned and communicated to users, it's not an incident—nothing is going on that requires an immediate, organized response. But usually, we'll be referring to unexpected outages caused by unanticipated failures. Most unexpected outages are incidents, or become incidents.

Incidents could result in confused customers. They could also cause losses in revenue, damaged data, breaches in security, and more, and these things can also impact your customers. When customers feel the impact of an incident, it might chip away at their trust in you as a provider. Therefore, you want to avoid having "too many" incidents or incidents that are "too severe" in order to keep your customers happy; otherwise, they will leave.

Having many incidents can also impact your incident responders, since handling incidents can be stressful. It can be challenging and expensive to find site reliability engineers (SREs) with the right mix of skills to respond to incidents, so you don't want to burn them out by designating them solely to incident response. Instead, you want to provide them with opportunities to grow their skills through proactive incident mitigation as well. We discuss this further later in this report, along with ways to reduce stress and improve the health of your on-call shifts.

[1] An incident is defined as an unplanned interruption or reduction in quality of an IT service (a service interruption). *ITIL_Glossary: Incident.*

Not Everything Is an Incident

Differentiating between *incidents* and *outages* is important. It's also important to differentiate between *metrics*, *alerts*, and *incidents*. How do you categorize *metrics* versus *alerts*, and *alerts* versus *incidents*? Not every metric becomes an alert, and not every alert is an incident. To help you understand the meaning of these terms, we'll start by discussing the role of monitoring and alerts to help maintain system health.

Monitoring

The most common way you keep watch over the health of your system is through monitoring. *Monitoring*,[2] as defined by the *Google SRE Book*, means collecting, processing, aggregating, and displaying real-time quantitative data about a system, such as query counts and types, error counts and types, processing times, and server lifetimes. Monitoring is a type of measurement.

When it comes to measurement, we suggest taking a customer-centric approach in regard to crafting both service-level objectives (SLOs; discussed in more detail in "Reducing the Impact of Incidents" on page 26) and the customer experience. This means collecting metrics that are good indications of the customer experience, and collecting a variety of measures, such as black box, infrastructure, client, and application metrics, wherever possible. Measuring the same values using different methods ensures redundancy and accuracy, since different measurement methods have different advantages. Customer-centric dashboards can also serve as good indications of the customer experience and are vital for troubleshooting and debugging incidents.

It's also important that your focus is on measuring reliability and the impact on your users, instead of on measuring the number of incidents that have been declared. If you focus on the latter, people will hesitate to declare an incident for fear of being penalized. This can lead to late incident declarations, which are problematic not only in terms of loss of time and loss of captured data, but also because

[2] Rob Ewaschuk, "Monitoring Distributed Systems", in *Site Reliability Engineering*, ed. Betsy Beyer, Chris Jones, Niall Richard Murphy, and Jennifer Petoff (Sebastopol, CA: O'Reilly Media, 2016).

an incident management protocol does not work well retroactively. Therefore, it's better to declare an incident and close it afterward than to open an incident retroactivity.

In that regard, people sometimes use the terms *reliability* and *availability* interchangeably, but reliability is more than just "service availability," especially in complex distributed systems. *Reliability* is the ability to provide a consistent level of service at scale. It includes different aspects, such as availability, latency, and accuracy. This can (and should) translate differently in different services. For example, does reliability mean the same for YouTube and Google Search? Depending on your service, your users' expectations will be different, and reliability can mean different things.

As a rule of thumb, a system is more reliable if it has fewer, shorter, and smaller outages. Therefore, what it all comes down to is the amount of downtime your users are willing to tolerate. As you take a customer-centric approach, the user defines your reliability. Consequently, you need to measure the user experience as closely as possible. (We discuss this in more detail in "Reducing the Impact of Incidents" on page 26.)

Alerting

We've discussed monitoring for system health. Now let's talk about the key component of monitoring: *alerting*. When monitoring identifies something that is not behaving as expected, it sends a signal that something is not right. That signal is an alert. An *alert* can mean one of two things: something is broken and somebody needs to fix it; or something might break soon, so somebody should take a look. The sense of *urgency*—that is, when an action needs to be taken—should direct you to choose how to respond. If an *immediate* (human) action is necessary, you should send a *page*. If a human action is required in the next several hours, you should send an *alert*. If no action is needed—that is, the information is needed in *pull* mode, such as for analysis or troubleshooting—the information remains in the form of *metrics* or *logs*.

Note that the alerting method can be different depending on the organization's preferences—for example, it can be visible in a dashboard or presented in the form of a ticket. At Google, it's usually the latter; a "bug" with different priorities is opened in the Google Issue Tracker by the monitoring system, which is our form of *ticketing*.

Now that you know the basics, let's take a deeper dive into alerting by discussing *actionable alerts*.

The Importance of Actionable Alerts

As we noted, an alert can trigger when a particular condition is met. You must be careful, however, to only alert on things that you care about *and* that are actionable. Consider the following scenario: as the active on-caller, you are paged at 2 a.m. because *QPS has increased by 300% in the past 5 minutes.* Perhaps this is a bursty service—there are periods of steady traffic, but then a large client comes and issues thousands of queries for an extended period of time.

What was the value in getting you out of bed in the middle of the night for this? There was no value, really. This alert was not actionable. As long as the service was not at risk of falling over, there was no reason to get anybody out of bed. Looking at historical data for your service would show that the service needs to be able to handle such traffic spikes, but the spikes themselves are not problematic and should not have generated any alerts.

Now let's consider a more nuanced (yet much more common) version of the actionable alerting problem. Your company requires making nightly backups of your production database, so you set up a cronjob that runs every four hours to make those backups. One of those runs failed because of a transient error—the replica serving the backup had a hardware failure, and was automatically taken out of serving mode by the load balancer—and consequent runs of the backup completed successfully. A ticket is subsequently created as a result of the failed run.

Creating a ticket because of one failed backup run is unnecessary. This would only result in noise, since the system recovered itself without human interaction.

These scenarios happen often. And although they end by simply closing the ticket with a "This was fine by the time I got to it" message, this behavior is problematic, for a few reasons:

Toil
: Someone had to spend time looking at the ticket, looking at graphs/reports, and deciding that they didn't need to do anything.

Alert fatigue
: If 95% of the "Database backups failed" alerts are simply closed, there's a much higher risk that an *actual* problem will go unnoticed.

As discussed earlier, an incident is an issue with particular characteristics. An alert is merely an indicator that can be used to signal that an incident is underway. You can have many alerts with no incidents. While this is an unfortunate situation, it doesn't mean you need to invoke formal incident management techniques; perhaps this is a planned maintenance event and you were expecting to receive these alerts as part of the maintenance process.

You can also have an incident without any alerts—maybe you were briefed by the security team that they suspect there's been a breach of your production systems; your team didn't have any alerts of their own that triggered for this particular condition.

More practically speaking, there are differences in how humans *perceive* alerts versus incidents:

- It's much more stressful to do formal incident management as opposed to simply fixing an alert.
- Less experienced responders are less likely to invoke an incident than more experienced responders.
- Incidents are much more likely to require additional team resources, so nonresponders can more easily gauge whether they need to start looking at the active issue sooner rather than later.

This applies not just within your team. In fact, it applies across the entire organization.

You typically have many more alerts than incidents. It's useful to get basic metrics around alerts (e.g., how many alerts there are per quarter), but incidents deserve taking a closer look (e.g., you had five major incidents last quarter, and they were all because of a new feature being rolled out which didn't have enough testing in pre-prod). You don't want to pollute these reports with all the alerts that you received. Consider the audience—alert metrics are primarily useful to the team, but incident reports will probably be read by higher-ups and should be scoped accordingly.

Hopefully, this clarifies when you can more confidently say, “This is not an incident.” However, this statement creates a dichotomy: if some things aren’t incidents, that means some things *are* incidents. How do you handle those? We’ll look at that in the next section.

The Incident Management Lifecycle

Optimal incident management doesn’t simply mean incidents are managed as quickly as possible. *Good incident management means paying attention to the whole lifecycle of an incident.* In this section, we discuss a programmatic approach to incident management. Think about incidents as a continuous risk existing in your system. The process of dealing with such risks is called the *incident management lifecycle*. The incident management lifecycle encompasses all of the necessary activities to prepare for, respond to, recover from, and mitigate incidents. This is an ongoing cost of an operational service.

By *lifecycle*, we mean every stage of an incident’s existence. These stages are shown in Figure 1-1 and described as follows:

Preparedness
: This encompasses all the actions a company or team takes to prepare for the occurrence of an incident. This can include safety measures on engineering (code reviews or rollout processes), incident management training, and experiments or testing exercises that are conducted to identify errors. This also includes setting up any monitoring or alerting.

Response
: This is what happens when the trigger causes the root cause of the hazard to become an issue. It involves responding to an alert, deciding whether the issue is an incident, and communicating about the incident to impacted individuals.

Mitigation and Recovery
: This is the set of actions that allow a system to restore itself to a functional state. These include the urgent mitigations needed in order to avoid impact or prevent growth in impact severity. Recovery includes the systems analysis and reflection involved in conducting a postmortem. A postmortem is a written record

of an incident, and it includes the actions taken, impact, root causes, and follow-up actions needed to prevent recurrence and/or reduce future impact.

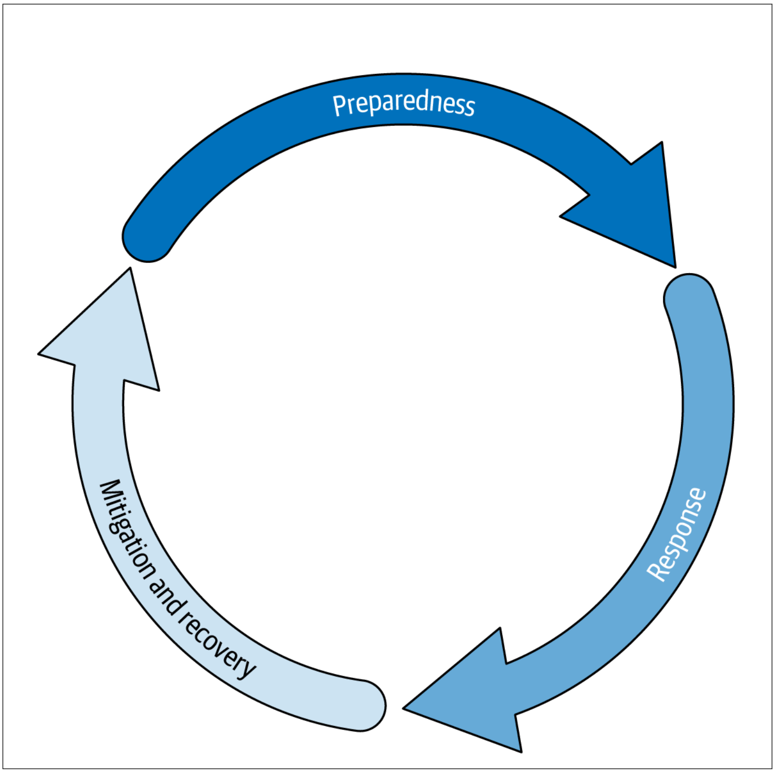


Figure 1-1. The incident management lifecycle

Once the recovery phase closes, you're thrust back into the preparedness phase. Depending on the size of your stack, it's possible that all of these phases occur simultaneously—but to be sure, you can expect at least one phase to always occur.

CHAPTER 2

Practicing Incident Response Readiness (Preparedness)

We talked about the stages of managing an incident and the incident management lifecycle. Now let's discuss how to practice incident management so that you can be ready when a real incident strikes.

Disaster Role-Playing and Incident Response Exercises

There is value in testing and practicing incident response readiness in order to increase resilience. We recommend implementing disaster role-playing in your team to train for incident response. At Google, we often refer to this as *Wheel of Misfortune*.[1] One way to do this is to re-create scenarios from real production incidents you encountered in the past.

There are tangible benefits to running regular incident response exercises. In the earlier days of Google's Disaster Resilience Testing (DiRT) program, there were tests deemed too risky to be executed. Over the years, by focusing on the areas exposed by those too-risky-to-run tests, many of these risks have been addressed so thoroughly that the tests are now automated and considered uninteresting.

[1] See Chapter 28 of *Site Reliability Engineering*.

Getting to that point wasn't immediate or painless—it took time and a lot of effort from several teams to get there—but we've been able to reduce significant risks in the global system to "just another automated test that runs periodically."[2]

Regular Testing

There are tangible benefits to regular testing. For years, Google has been running DiRT tests to find and remediate problems with our production systems. As teams have been testing their services, there has been a decrease in the number of high-risk tests. This is actually a good sign—teams have made their systems much more resilient, to the point where finding weaknesses is becoming harder.

Tests-gone-wrong—situations in which tests have failed for some reason—have also become much less frequent. Even when they happen, these systems tend to fail in ways that were predicted, and by extension, were remediated quickly. Responders are now much more comfortable activating emergency procedures, keeping cool under pressure, and as a result of these tests, writing fewer postmortems. Years of hard work have paid off—the mentality has evolved from "disaster testing is something that happens to me" to "disaster testing is something we all do together."

Nuanced Testing and Automation

Testing is slowly shifting from fixing purely technical problems (e.g., "Do we know how to restore from a totally corrupt database?") to a much more nuanced "Let's fix *processes*" set of challenges.

Technical tests are easier to discuss and automate: it all boils down to writing some code to execute a series of commands and check an expected response. It is more difficult to find problematic processes—such as "only one person is authorized to approve this, but they're not responding to their phone/email"—especially ones that are not executed often.

[2] Marc Alvidrez, "The Global Chubby Planned Outage" in *Site Reliability Engineering*.

Preparing Responders

Running incident response tests—even if only theoretical—can help identify such processes, assign a probability and risk factor, and instill confidence in responders. Even if a particular test did not go as planned, you will gain better visibility into *where* the weaknesses of your incident response processes are. Responders will also be better prepared—technically, mentally, and emotionally—for whenever they have to deal with a real incident.

Emotional preparedness should not be underestimated. As noted earlier, incident management can put responders under significant amounts of stress. Stress can lead to poor responses, such as oversights, slower responses, and clouded judgment. It also can cause anxiety, exhaustion, high blood pressure, and poor sleep, among other health issues.

Being able to run incident response tests can better prepare individuals to not just *reduce* these adverse effects, but more importantly, *identify* them so that they can take corrective actions—such as asking for additional help, taking a break, or even handing off an incident completely. Managers and others in leadership positions should also constantly be on the lookout for signs of stress/fatigue/burnout in responders and assist them whenever possible.

Writing Incident Response Tests

A good starting point for writing incident response tests is to look at recent incidents. At Google, we ask these standard questions on every postmortem:

- What went wrong?
- What went right?
- Where did we get lucky?

Start by looking at what went wrong, since that's clearly an area that needs improvement. These tend to be concrete problems that are easy to fix—for example, *your monitoring picked up an issue but didn't page anybody*. Once you've identified and fixed the problem, you need to test the fix. This point cannot be overemphasized: merely fixing an issue is not enough; it's possible that the fix may be incomplete, or the fix has caused a regression somewhere else. When testing for correctness, start with small, relatively simple tests.

As confidence in the process increases, you can start looking at more complex issues, including those that aren't entirely technical in nature (i.e., human processes).

Once these smaller-scoped tests have been underway for some time, start looking at the "where did we get lucky?" items. Oftentimes, these problems are much more subtle, and addressing them might not be trivial. Once again, start small. Break down these issues into smaller, easier-to-address items.

These tests should have a slow but steady cadence—you don't want to drown teams in these tests, but you don't want to lose momentum either. It is also easier to justify a one-hour test every four weeks (as an example) rather than spending 10% of your operational budget on these tests. As these procedures evolve and the value of running these tests becomes clearer, you'll find a natural pattern for how often these tests should be conducted and how thorough they should be.

CHAPTER 3

Scaling Incident Management (Response)

We've discussed practicing incident response readiness by conducting incident response exercises, role-playing, and running regular tests. These tactics help you get ready for when a real incident occurs and you start managing the incident (see "Establish an organized incident response procedure" on page 31). But how do you manage incidents once your organization starts to grow? In this section, we discuss how to scale incident management.

At Google, we're set up to provide optimal incident management coverage for all systems. Google's gotten really big. To serve more than 2 trillion queries per year, we leverage a lot of data centers, at least a million computers, and more than 80,000 employees. All this activity is routed through a massive and highly interconnected system-of-systems (SoS), critically reliant on its technical stack to be in active production. The support of this technical stack implies that appropriate personnel are reliably available in order to troubleshoot and correct issues as they arise. These are the responders in our Site Reliability Engineering organization; they provide incident management coverage for systems and respond when an incident occurs.

Component Responders

Within the Site Reliability Engineering organization, we also have *component responders*, who are incident responders on call for one component or system within Google's overall technical infrastructure (Figure 3-1).

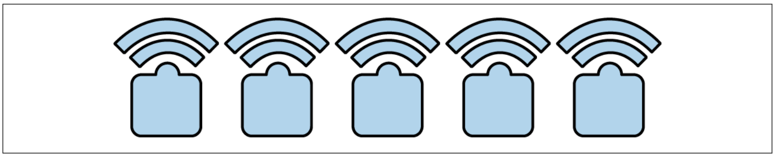


Figure 3-1. Component responders

Component responders are single-system experts who are well versed in the problem space, are expert troubleshooters, and are practiced in implementing mitigation strategies during a crisis. They have continual access to the tools/systems required to perform emergency response, to handle stress well, and to think clearly during a crisis.

Individual component responders have a limited area of responsibility; this allows them to maintain in-depth knowledge of their area and the systems that can affect it. These responders are the first line of defense against failures cascading from one component through the whole stack. These individual components are smaller than the overall SoS stack, as we will discuss in "System-of-System (SoS) Responders" on page 17, and usually have clear and distinct system boundaries. As a result, appropriate monitoring and alerting can be set up so that component responders remain knowledgeable about their system's failure modes.

When the scope of a technical stack grows beyond one person's capacity to understand and maintain state, we split up the technical stack such that multiple responders can each provide coverage on a single component of the whole stack. As time passes, those components grow more complex and become further divided. By maintaining a limited scope, primary responders can resolve smaller-scoped issues at any given time. There is, however, the risk of remaining ignorant of production issues spanning multiple components, between system-to-system boundaries (see the next section), or of not providing component responders with sufficient support if an issue proves to be beyond their expertise.

For example, say an underlying fault is cascading through a notable fraction of the technical stack. This cascade is happening faster than humans can self-organize. During an incident with broad impact, we quickly achieve a situation where every component team has been paged, has assigned responders, and is managing their own state. These component teams are working in parallel, but there's a possibility that none of those responders was aware of the others (Figure 3-2). One of those responder's incidents may be the root cause, while the rest describe consequences. But which one?

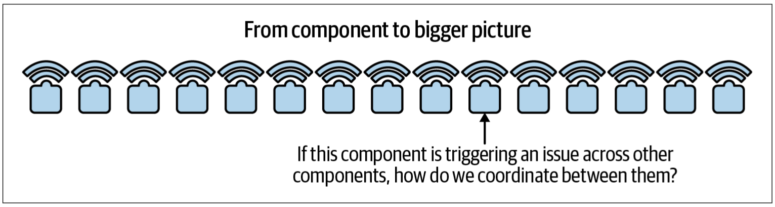

From component to bigger picture
If this component is triggering an issue across other components, how do we coordinate between them?

Figure 3-2. From components to the bigger picture

With a sufficiently large and complex technical stack, it becomes highly unlikely for a single primary responder to be able to drive mitigation and maintain state on all their dependencies and their dependents. To mitigate this risk, we built a structure of secondary responders beyond our brilliant group of component responders. We at Google call these secondary responders system-of-system responders, which we discuss next.

System-of-System (SoS) Responders

System-of-system (SoS) responders are incident responders who are on call to support incidents that span multiple component systems, incidents that fall between system boundaries, or anything that gets messy. SoS responders are appropriately trained, politically situated, and empowered to lead an organized, coordinated response. These responders are the second line of defense, which is more holistically focused as well, and they provide key advantages when responding to distributed computing failures (Figure 3-3).

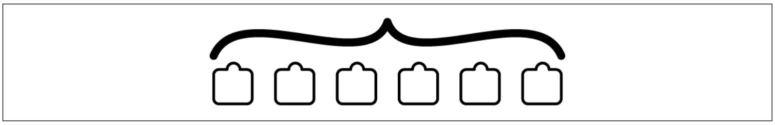


Figure 3-3. SoS responders

We regard SoS responders as multisystem incident managers, skilled generalists, and holistically focused; they have expertise in handling a subset of incidents where we need a wider perspective. Often, these incidents require the involvement of multiple teams; an example would be a major SoS outage that brings down many services. These incidents may cause or have caused downstream failures, and have the potential to expand beyond service boundaries. In addition, they may be customer-impacting incidents that have been ongoing for 30 minutes or more with no sign of resolution.

SoS responders are well situated to respond to these types of far-reaching incidents because they know how to organize others and take command of complex situations. They also know how to diagnose systemic behaviors and identify root causes, and are focused on scaling the response and communicating widely about the incident.

At Google, we have two types of SoS responders. While each type has a distinct function, they often interoperate:

Product-focused incident response teams (IRTs)

These are incident response teams that are in place to protect the reliability of a specific product area. Examples include the Ads IRT and the YouTube IRT. Not every product area is required to have an incident response team, but it's helpful as products continually roll out new features, grow in complexity, and have similarly scented technical debt. These teams are folks who won't know every detail of a product's stack, but they are aware of a product's holistic operations and dependencies.

Technical incident response team (Tech IRT)

This is our broadest-focused incident response team. This team focuses on incidents that span across products, have ambiguous ownership, or have an unclear and pervasive root cause. Tech IRT is our last line of defense. Members of Tech IRT are tenured Googlers who have worked as component responders on at least two different teams. They have a broad understanding of how things work, and most importantly, they have excellent incident management skills.

Members of Tech IRT continue to work for their home components, but serve shifts on a global, 24-7 pager rotation. And they can still work during these kinds of major emergencies because they practice this niche skill set. A lot.

Tech IRT members undergo two weeks of production training twice a year, which covers the deep details of how things work (and fail). They also need to give quarterly demonstrations on their ability to work productively using emergency tools.

Figure 3-4 depicts the incident response organizational architecture at Google. With each additional level of this architecture, there is an additional level of abstraction from the intricate details of a product's daily functioning. Each role is equally important—each subsequent level of this pyramid experiences less pager load. If a component responder cannot resolve an issue, and it threatens the stability of the product, there's someone they can escalate it to: a product-focused IRT.

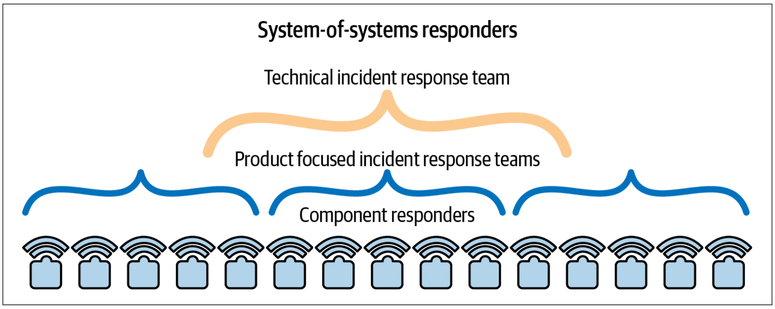

System-of-systems responders
Technical incident response team
Product focused incident response teams
Component responders

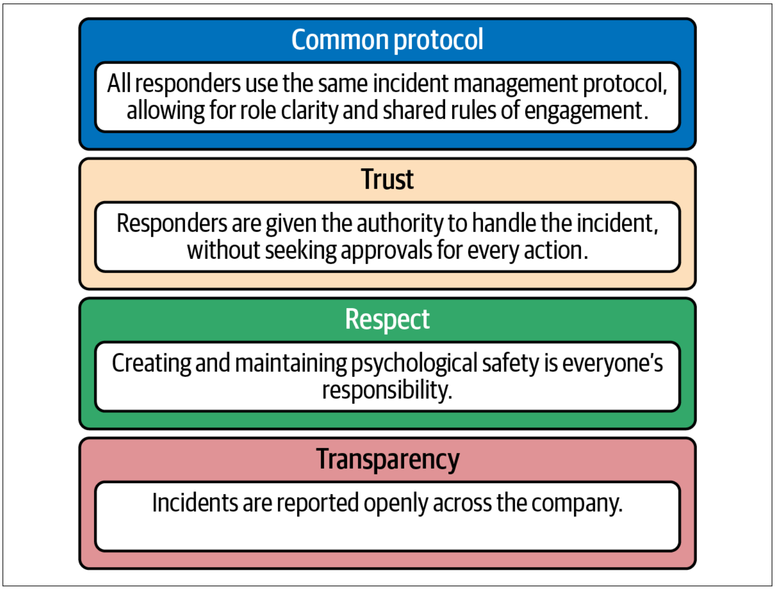
Figure 3-4. Incident response organizational architecture

If an issue is threatening multiple products, or if an issue can be more quickly mitigated through a solution on shared infrastructure, Tech IRT will be activated, operating on the broadest scope as a point of escalation for everything below it.

Now, what enables this organizational architecture to function seamlessly? The answer is *a common protocol*, *trust*, *respect*, and *transparency*. Let's look at these next.

Incident Response Organizational Structure

There are four characteristics of a successful incident response organization: a common protocol, trust, respect, and transparency (see Figure 3-5).



Common protocol

All responders use the same incident management protocol, allowing for role clarity and shared rules of engagement.

Trust

Responders are given the authority to handle the incident, without seeking approvals for every action.

Respect

Creating and maintaining psychological safety is everyone's responsibility.

Transparency

Incidents are reported openly across the company.

Figure 3-5. Characteristics of a successful incident response organization

Common Protocol

At Google, we widely leverage an internal variant of the FEMA Incident Command System (ICS) in which incident responders have defined roles, such as incident command, scribe, or communications. By using a shared, clearly defined process, we build positive emergency response habits, including maintenance of active state, a clear chain of command, and an overall reduction of stress. Everyone understands the handoff process, including whom to hand off to, to ensure effective transfer of knowledge. A chess player can't drop a bishop on a mah-jongg table and guarantee that everyone knows what to do with it—in urgent situations, it's important that all players are playing the same game.

Trust

During an incident, incident commanders need to wield authority. They need to direct others, organize chaotic energy, and make judgments about an appropriate course of action. For many organizations, aligning authority with operational duty is a challenging concept, but our standard operating procedure avoids the convention that upper-level business directors are the only individuals who should have the authority to make service-altering judgment calls: we give this authority to the subject matter experts who have context and state.

Respect

It's important to make sure all responders feel comfortable escalating the situation when they feel it is necessary. If responders were subject to scrutiny, criticism, or perceptions of ineptitude for escalating incidents, they might not escalate when it would be appropriate to do so. Beyond common courtesy, we have to trust that individuals make the best decisions possible, given the information they have in front of them. If something goes awry, the critical issue isn't to make someone feel bad for it—it's more important to figure out how to create more accurate and actionable information to make sure things don't go wrong in the future. Part of this comes during the postmortem process in which Google maintains a strict policy of blamelessness (more on this shortly).

Transparency

We don't silo. When an incident occurs, the details are made available to everyone. Escalation and interoperability can't happen if we're prohibited from accessing information about incidents—and the postmortems we write after incidents have been resolved are shared company-wide in our weekly newsletter. We encourage the type of cross-learning that results from reading about incidents occurring on other teams and in other product areas.

Managing Risk

In addition to the characteristics of the incident response organizational structure, you should also think about how to manage risk. The time from identifying to resolving an incident should be no more than three days. As we said earlier, incident management is expensive in time and personnel. Keeping yourself in a state of active incident management for a long period of time will cause fatigue and burnout, which may convince you to start refreshing your resume. Incidents are issues that have been escalated and that require an immediate, organized response. This state of immediacy isn't a natural state—humans aren't supposed to have their medulla oblongata stimulated for that long, nor are they supposed to have that much cortisol pumping through their bodies for that long. If prehistoric people were constantly hunting or being hunted by saber-toothed tigers and never felt safe or had time to rest, we would have evolved much differently. If you expect to spend that much time in fight-or-flight mode, it's natural to expect that this situation will eventually lead to continuous turnover on your team.

The Function of Incident Management and Risk

To minimize time spent in incident management mode, it's important to recognize the function of incident management and risk. Incident management is a *short-term* exercise intended to rapidly correct a risky situation. The severity of an incident breaks down into a few simple categories. At Google, we have quantified these appropriately for our organization's products (see Table 3-1).

Table 3-1. Severity definitions

Severity	Definition	Litmus Test
Huge	A major user-facing outage that generates bad press *OR* that results in a massive impact on revenue for Google or identified Google customers. An *internal* productivity outage would only be considered huge if there were visible external consequences that caused, for example, a negative press cycle.	Could or did damage the Alphabet/Google brand and business.
Major	An outage that is visible to users but does not cause lasting damage to Google services or identified customers, *OR* a sizable loss in revenue for Google or its customers, *OR* 50% or more of Googlers significantly impacted	Recurring, unmitigated future incidents of this nature could or will damage the Alphabet/Google brand and business.
Medium	Anywhere from a near miss to a huge/major outage. A significant number of internal users are significantly impacted. Workarounds existed and were known to users (mitigating the impact).	Recurring, unmitigated future incidents of this nature will likely lead to increasing instability over time and greater costs in production maintenance.
Minor	External users may not have even noticed the outage. Internal users were inconvenienced. The result was unexpected sloshing of traffic between entities (network, data center, instances).	Recurring, unmitigated future incidents of this nature are unlikely to lead to increasing instability over time but represent normal operating conditions.

Severity	Definition	Litmus Test
Negligible, Trivial	The incident was not visible to users in any way and had little to no real impact on production, but valuable lessons were learned and some follow-up action items may need to be tracked at a low priority.	Recurring, unmitigated future incidents of this nature would not be considered a process breakdown.
Test, Ignored	This wasn't even an incident. Go do a new thing.	False alarm.

The size of an incident maps roughly to the "riskiness" of the situation (root cause/trigger/impact). Incident management mitigates the short-term effects of the impact to buy some time for the folks in power at the organization to determine what should happen next. Since the incident has determined that "X is a problem!" and "someone should do something!" the incident response is intended to ensure that any potential short-term impact is mitigated in order to make some long-term decisions. This doesn't mean incident management should continue until both the short-term and long-term impacts are avoided. With a sizable tech stack, or depending on the volume of tech debt that you might have, it could take months or even years to fully rectify the root cause/trigger conditions that erupted. The incident should only remain "open," with active incident management taking place, until the short-term impact is mitigated. In a hospital setting, the equivalent would be to evaluate a bleeding patient for imminent risk and then to stop the bleeding.

But what comes next? In a hospital, the next step is to determine what caused the bleeding and how to prevent it from recurring. Maybe this involves helping the patient develop a long-term saber-toothed tiger avoidance plan, or maybe it involves treating the skin disease that caused the open wound. Either way, once the immediate danger has been averted, longer-term plans, including a short period of round-the-clock support, if necessary, are put in place to keep the patient safe and prevent the bleeding from happening again. Similarly, in your tech stack, once the immediate danger has been averted it's time to pivot and work out longer-term actions.

In incident management, you can often re-create the timeline of an incident in minutes. If you're dealing with an active, urgent issue, every minute might reflect time in which users are being harmed or revenue is being lost. Because every minute matters, incident management creates a lot of pressure on incident managers—and, as noted earlier in this section, that's not a long-term positive experience. When you’re working on the longer-term aftereffects of an incident (resolving the root cause or the trigger), you’re ideally out of the way of immediate user harm or significant profit loss. Cool. This leaves you with some high-priority work that still needs to be executed immediately, but it doesn’t need to be executed in the same way that incidents are managed. The timeline might be better measured in days or weeks, and not in the recommended incident timeline of no more than three days that we mentioned earlier. Don’t stay in fight-or-flight mode after you need to. Close your incident; move on to recovery.

CHAPTER 4

Mitigation and Recovery

We've talked about scaling incident management, and using component responders and SoS responders to help manage incidents as your company scales. We've also covered the characteristics of a successful incident response organization, and discussed managing risk and preventing on-call burnout. Here, we talk about recovery after an incident has occurred. We'll start by focusing on urgent mitigations.

Urgent Mitigations

Previously, we encouraged you to "stop the bleeding" during a service incident. We also established that recovery includes the urgent mitigations[1] needed in order to avoid impact or prevent growth in impact severity. Let's touch on what that means and some ways to make mitigation easier during urgent circumstances.

Imagine that your service is having a bad time. The outage has begun, it's been detected, it's causing user impact, and you're at the helm. Your first priority should always be to stop or lessen the user impact, *not* to figure out what's causing the issue. Imagine you're in a house and the roof begins to leak. The first thing you're likely to do is place a bucket under the dripping water to prevent further water damage, *before* you grab your roofing supplies and head upstairs to figure out what's causing the leak. (As we'll find out later, if the

[1] Recommended reading: "Generic mitigations" by Jennifer Mace.

roofing failures are the *root cause*, the rain is the *trigger*.) The bucket reduces the impact until the roof is fixed and the sky clears. To stop or lessen user impact during a service breakage, you'll want to have some buckets ready to go. We refer to these metaphorical buckets as *generic mitigations*.

A generic mitigation is an action that you can take to reduce the impact of a wide variety of outages while you're figuring out what needs to be fixed.

The mitigations that are most applicable to your service vary, depending on the pathways by which your users can be impacted. Some of the basic building blocks are the ability to roll back a binary, drain or relocate traffic, and add capacity. These Band-Aids are intended to buy you and your service time so that you can figure out a meaningful fix which can fully resolve the underlying issues. In other words, they fix the *symptoms* of the outage rather than the *causes*. You shouldn't have to fully understand your outage to use a generic mitigation.

Consider doing the research and investing in the development of some quick-fix buttons (the metaphorical buckets). Remember that a bucket might be a simple tool, but it can still be used improperly. Therefore, to use generic mitigations correctly, it's important to practice them during regular resilience testing.

Reducing the Impact of Incidents

Besides generic mitigations, which you first use to mitigate an urgent situation or an incident, you'll need to think about how to reduce the impact of incidents in the long run. *Incidents* is an internal term. In reality, your customers don't really care about incidents or the number of incidents—what they *do* really care about is reliability. To meet your users' expectations and achieve the desired level of reliability, you need to design and run reliable systems. To do that you need to align your actions for each stage in the incident management lifecycle mentioned previously: preparedness, response, and recovery. Think about the things you can do before, during, and after the incident to improve your systems.

While it's very difficult to measure customer trust, there are some proxies you can use to measure how well you're providing a reliable customer experience. We call the measurement of customer experience a service-level indicator (SLI). An SLI tells you how well your service is doing at any moment in time. Is it performing acceptably, or not?

For this scope, the *customer* can be an end user, a human or system (such as an API), or another internal service. The internal service is like a core service serving another internal service, which serves the end user. In that regard, you can be as reliable as your critical dependencies (i.e., a hard dependency or a dependency that cannot be mitigated—if it fails, you fail). This means that if your customer-facing services depend on internal services, those services need to provide a *higher* level of reliability to provide the needed reliability level to the customer.

The *reliability target* for an SLI is called a service-level objective (SLO). An SLO aggregates the target over time: it says that during a certain period, this is your target and this is how well you are performing against it (often measured as a percentage).[2]

Most of you are probably familiar with service-level agreements (SLAs). An SLA defines what you've promised to provide your customers; that is, what you're willing to do (e.g., refund money) if you fail to meet your objectives. To achieve this, you need your SLOs—your targets—to be more restrictive than your SLAs.[3]

The tool we use to examine and measure our users' happiness is called the *user journey*. User journeys are textual statements written to represent the user perspective. User journeys explore how your users are interacting with your service to achieve a set of goals. The most important user journey is the *critical user journey* (CUJ).[4]

Once you've defined the targets that are important to you and your users or customers, you can start to think about what happens when you fail to meet those targets.

[2] See Chapter 4, "Service Level Objectives," in *Site Reliability Engineering* (O'Reilly).

[3] See Adrian Hilton's post "SRE Fundamentals 2021: SLIs vs SLAs vs SLOs", May 7, 2021.

[4] See Chapter 2, "Implementing SLOs," in *The Site Reliability Workbook* (O'Reilly).

Calculating the Impact of Incidents

Incidents impact the reliability target. They are affected by the number of failures you have, the length and the blast radius, and the "size" of these failures. So, to reduce the impact of an incident, you first need to understand what you can do to reduce the impact. Let's look at how to quantify and measure the impact of an incident.

Figure 4-1 demonstrates that to measure the impact, you calculate the time that you are not reliable. This is the time it takes for you to detect that there is an impact, plus the time it takes to repair (mitigate) it. You then multiply this by the number of incidents, which is determined by the frequency of incidents.

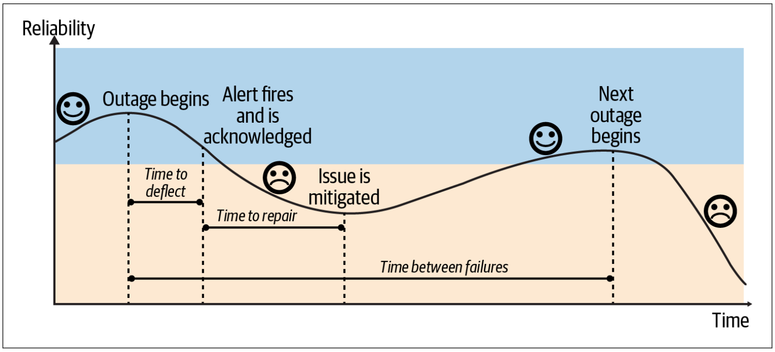


Figure 4-1. Outage lifecycle

The key metrics are time to detect, time to repair, and time between failures:

- Time to detect (TTD) is the amount of time from when an outage occurs to some human being notified or alerted that an issue is occurring.
- Time to repair (TTR) begins when someone is alerted to the problem and ends when the problem has been mitigated. The key word here is *mitigated*! This doesn't mean the time it took you to submit code to fix the problem. It's the time it took the responder to mitigate the customer impact; for example, by shifting traffic to another region.
- Time between failures (TBF) is the time from the beginning of one incident to the beginning of the next incident of the same type.

Reducing customer impact means reducing the four axes in the following equation—time to detect, time to repair, time between failures, and impact.

To reduce the impact of incidents and enable systems to recover to a known state, you need a combination of technology and "human" aspects, such as processes and enablement. At Google, we found that once a human is involved, the outage will last at least 20 to 30 minutes. In general, automation and self-healing systems are a great strategy, since both help reduce the time to detect and time to repair.

$$\mathit{Unreliability}\ \Sigma \frac{TTD+TTR}{TBF} \times \mathit{Impact}$$

It's important to note that you should also be mindful of the method you use. Simply decreasing your alerting threshold can lead to false positives and noise, and relying too heavily on implementing quick fixes using automation might reduce the time to repair but lead to ignoring the underlying issue. In the next section, we share several strategies that can help reduce the time to detect, time to repair, and frequency of your incidents in a more strategic way.

Reducing the Time to Detect

One way to reduce the impact of incidents is to reduce the time to detect the incident (Figure 4-2). As part of drafting your SLO (your reliability target), you perform a risk analysis and figure out what you need to prioritize, and then you identify what can prevent you from achieving your SLO; this can also help you reduce the time to detect an incident. In addition, you can do the following to minimize the time to detect:

- *Align your SLIs*, your indicators for customer happiness, as close as you can to the expectations of your users, which can be real people or other services. Furthermore, align your alerts with your SLOs (i.e., your targets), and review them periodically to make sure they still represent your users' happiness.
- *Use fresh signal data*. By this, we mean that you should measure your quality alerts using different measurement strategies, as we discussed earlier. It's important to choose what works best to get the data: streams, logs, or batch processing. In that regard,

it's also important to find the right balance between alerting too quickly, which can cause noise, and alerting too slowly, which may impact your users. (Note that noisy alerts are one of the most common complaints you hear from Ops teams, be they traditional DevOps teams or SREs.)

- *Use effective alerts* to avoid alert fatigue. Use pages when you need immediate action. Only the *right responders*—the specific team and owners—should get the alerts. (Note that another common complaint is having alerts that are not actionable.)

  However, a follow-up question to this is: "If you page only on things that require immediate action, what do you do with the rest of the issues?" One solution is to have different tools and platforms for different reasons. Maybe the "right platform" is a ticketing system or a dashboard, or maybe you only need the metric for troubleshooting and debugging in a "pull" mode.

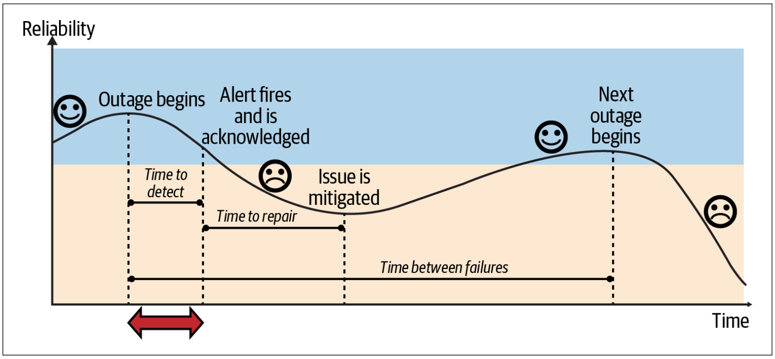


Figure 4-2. Outage lifecycle: time to detect

Reducing the Time to Repair

We discussed reducing the time to detect as one way to reduce the impact of incidents. Another way to reduce the impact is by reducing the time to repair (Figure 4-3). Reducing the time to repair is mostly about the "human side." Using incident management protocols and organizing an incident management response reduces the ambiguity of incident management and the time to mitigate the impact on your users. Beyond that, you want to train the responders, have clear procedures and playbooks, and reduce the stress around on-call. Let's look at these strategies in detail.

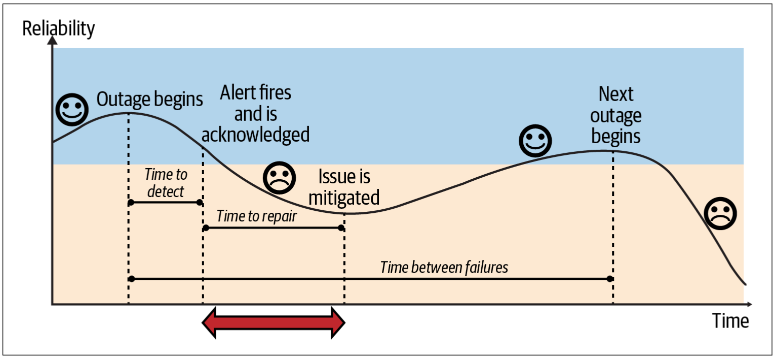


Figure 4-3. Outage lifecycle: time to repair

Train the responders

Unprepared on-callers lead to longer repair times. Consider conducting on-call training sessions on topics such as disaster recovery testing, or running the Wheel of Misfortune exercise we mentioned earlier. Another technique is a mentored ramp-up to on-call. Having on-callers work in pairs ("pair on call"), or having an apprenticeship where the mentee joins an experienced on-caller during their shifts ("shadowing"), can be helpful in growing confidence in new teammates. Remember that on-call can be stressful. Having clear incident management processes can reduce that stress as it eliminates any ambiguity and clarifies the actions that are needed.[5]

Establish an organized incident response procedure

There are some common problems regarding incident management. For example, lack of accountability, poor communications, missing hierarchy, and freelancing/heroes can result in longer resolution times, add additional stress for the on-callers and responders, and impact your customers. To address this, we recommend organizing a response by establishing a hierarchical structure with clear roles, tasks, and communication channels. This helps maintain a clear line of command and designates clearly defined roles.

At Google, we use IMAG (Incident Management at Google), a flexible framework based on the Incident Command System (ICS) used by firefighters and medics. IMAG teaches you how to organize an

[5] See Jesus Climent's post "Shrinking the Time to Mitigate Production Incidents—CRE Life Lessons", December 5, 2019.

emergency response by establishing a hierarchical structure with clear roles, tasks, and communication channels (Figure 4-4). It establishes a standard, consistent way to handle emergencies and organize an effective response.[6]

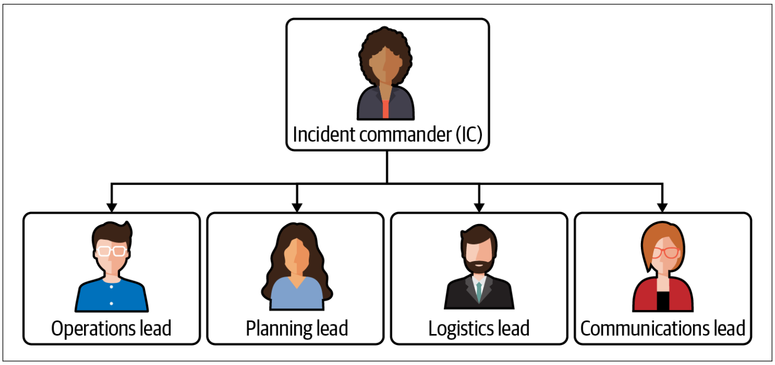


Figure 4-4. An example ICS hierarchy

The IMAG protocol provides a framework for those working to resolve an incident, enabling the emergency response team to be self-organized and efficient by ensuring *communication* between responders and relevant stakeholders, keeping *control* over the incident response, and helping *coordinate* the response effort. It asserts that the incident commander (IC) is responsible for coordinating the response and delegating responsibilities, while everyone else reports to the IC. Each person has a specific, defined role—for example, the operations lead is responsible for fixing the issue, and the communications lead is responsible for handling communication.

By using such a protocol, you can reduce ambiguity, make it clear that it's a team effort, and reduce the time to repair.[7]

Establish clear on-call policies and processes

We recommend documenting your incident response and on-call policies, as well as your emergency response processes, both during and after an outage. This includes a clear escalation path and the assignment of responsibilities during an outage. This reduces the ambiguity and stress associated with handling outages.

[6] See Chapter 9, "Incident Response," in *The Site Reliability Workbook*.

[7] See Chapter 14, "Managing Incidents," in *Site Reliability Engineering*.

Write useful runbooks/playbooks

Documentation is important, since it helps turn on-the-job experience into knowledge available to all teammates regardless of tenure. By prioritizing and setting time aside for documentation, as well as creating playbooks and policies that capture procedures, you can have teammates who readily recognize how an incident might present itself—a valuable advantage. Playbooks don't have to be robust at first; start simple to provide a clear starting point, and then iterate. A good rule of thumb is Google's *see it, fix it* approach (i.e., solve problems as you uncover them), and letting new team members update those playbooks as part of their onboarding.

Make playbook development a key postmortem action item, and recognize it as a positive team contribution from a performance management perspective. This usually requires leadership prioritization and allocating the necessary resources as part of the development sprint.

Reduce responder fatigue

As mentioned in Chapter 2, the mental cost of responder fatigue is well documented. Furthermore, if your responders are exhausted, this will affect their ability to resolve issues. You need to make sure shifts are balanced, and if they aren't, to use data to help you understand why and reduce the toil.[8]

Invest in data collection and observability

You want to be able to make decisions based on data, so a lack of monitoring or observability is an antipattern. If you cannot see, you won't know where you are going. Therefore, encourage a culture of measurement in the organization, collecting metrics that are close to the customer experience, and measure how well you are doing against your targets and your error budget burn rate so that you can react and adjust priorities. Also, measure the team's toil and review your SLIs and SLOs periodically.

You want to have as much quality data as you can. It's especially important to measure things as close to the customer experience as possible; this helps you troubleshoot and debug the problem. Collect

[8] See Eric Harvieux's post"Identifying and Tracking Toil Using SRE Principles", January 31, 2020.

application and business metrics, in order to have dashboards and visualization focused on the customer experience and critical user journeys. This means having dashboards that aim for a specific audience with specific goals in mind. A manager's view of SLOs is very different from a dashboard that needs to be used for troubleshooting an incident.

As you can see, there are several things you can do to reduce the time to repair and minimize the impact of incidents. Now let's look at increasing the time between failures as another way to reduce the impact of incidents.

Increasing the Time Between Failures

To increase the time between failures and reduce the number of failures, you can refactor the architecture and address the points of failure that were identified during risk analysis and process improvement (Figure 4-5). You can also do several additional things to increase the TBF.

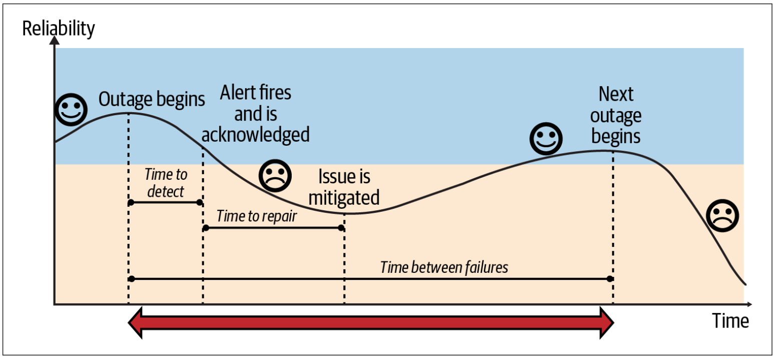


Figure 4-5. Outage lifecycle: time between failures

Avoid antipatterns

We mentioned several antipatterns throughout this report, including a lack of observability and having positive feedback loops, which can overload the system and can cause cascading issues like crashing. You want to avoid these antipatterns.

Spread risks

You should spread the risks by having redundancies, decoupling responsibilities, avoiding single points of failure and global changes, and adopting advanced deployment strategies. Consider progressive and canary rollouts over the course of hours, days, or weeks, which allow you to reduce risk and identify an issue *before* all your users are affected. Similarly, it's good to have automated testing, gradual rollouts, and automatic rollbacks to catch any issues early on. Find the issues before they find you; achieve this by practicing chaos engineering and introducing fault injection and automated disaster recovery testing such as DiRT (see Chapter 2).

Adopt dev practices

You can also adopt dev practices that foster a culture of quality, and create an integrated process of code review and robust testing which can be integrated into Continuous Integration/Continuous Delivery (CI/CD) pipelines. CI/CD saves engineering time and reduces customer impact, allowing you to deploy with confidence.

Design with reliability in mind

In SRE, we have a saying: "Hope is not a strategy." When it comes to failures, it's not a question of *if*, but *when*. Therefore, you design with reliability in mind from the very beginning, and have robust architectures that can accommodate failures. It's important to understand how you deal with failure by asking yourself the following questions:

- What type of failure is my system resilient to?
- Can it tolerate unexpected, single-instance failure or a reboot?
- How will it deal with zonal or regional failures?

After you are aware of the risks and their potential blast radius, you can move to risk mitigation (as you do during risk analysis). For example, to mitigate single-instance issues, you should use persistent disks and provision automation, and, of course, you should back up your data. To mitigate zonal and regional failures, you can have a variety of resources across regions and zones and implement load balancing. Another thing you can do is scale horizontally. For example, if you decouple your monolith to microservices, it's easier to scale them independently ("Do one thing and do it well"). Scaling horizontally can also mean scaling geographically, such as having multiple data centers to take advantage of elasticity. We recommend avoiding manual configuration and special hardware whenever possible.

Graceful degradation

It's important to implement graceful degradation methods in your architecture. Consider degradation as a strategy, like throttling and load shedding. Ask yourself, if I can't serve all users with all features, can I serve all users with minimal functionality? Can I throttle user traffic and drop expensive requests? Of course, what is considered an acceptable degradation is highly dependent on the service and user journey. There is a difference between returning *x* products and returning a checking account balance that is not updated. As a rule of thumb, however, degraded service is better than no service.[9]

Defense-in-depth

Defense-in-depth is another variation of how you build your system to deal with failures, or more correctly, to tolerate failures. If you rely on a system for configuration or other runtime information, ensure that you have a fallback, or a cached version that will continue to work when the dependency becomes unavailable.[10]

N+2 resources

Having N+2 resources is a minimum principle for achieving reliability in a distributed system. N+2 means you have N capacity to serve the requests at peak, and +2 instances to allow for one instance (of the complete system) to be unavailable due to unexpected failure and another instance to be unavailable due to planned upgrades. As we mentioned, you can only be as reliable as your critical dependencies are, so choose the right building blocks in your architecture. When building in the cloud, ensure the reliability levels of the services that you use and correlate them with your application targets. Be mindful of their scope (e.g., in Google Cloud Platform's building blocks [zonal, regional, global]). Remember, when it comes to design and architecture, addressing reliability issues during design

[9] For more on load shedding and graceful degradation, see Chapter 22, "Addressing Cascading Failures," in *Site Reliability Engineering*.

[10] See the Google Blog post by Ines Envid and Emil Kiner: "Google Cloud Networking in Depth: Three Defense-in-Depth Principles for Securing Your Environment", June 20, 2019.

reduces the cost later.[11] There is no one-size-fits-all solution; you should let your requirements guide you.

NALSD: Non-Abstract Large System Design

We can't talk about designing for reliability and SRE without touching on non-abstract, large system design. At Google, we found that addressing reliability issues during the design phase reduces future costs, and if we adapt an iterative style of system design and implementation, we can develop robust and scalable systems at a lower cost. We call this approach *non-abstract large system design*, or *NALSD*, and it describes the iterative process of system design that Google uses for production systems. You can learn more about it in Google's SRE Classroom.

Learn from failures

Finally, you can learn from failures in order to make tomorrow better (more on that in "Psychological Safety" on page 40). As we mentioned before, the tool for this is postmortems. Ensure that you have a consistent postmortem process that produces action items for bug fixes, mitigations, and documentation updates. Track postmortem action items the same as you would any other bug (if you are not doing so already), and prioritize postmortem work over "regular" work.[12] We discuss postmortems in more detail in the next section.

[11] See Chapter 12, "Introducing Non-Abstract Large System Design," in *The Site Reliability Workbook*.

[12] See the article for Google Research by Betsy Beyer, John Lunney, and Sue Lueder: "Postmortem Action Items: Plan the Work and Work the Plan".

CHAPTER 5

Postmortems and Beyond

In the previous chapter, we covered several things you can do to reduce customer impact, in terms of both technology and people, since both affect the time to detect, the time to mitigate/recover, and the time between failures. In this section, we talk about what happens after an incident has concluded: writing postmortems and using them as a powerful tool to analyze what went wrong and learn from mistakes.

After an incident has concluded, how do you know where to focus your efforts to minimize future incidents? To know what you should focus on, we recommend taking a data-driven approach (Figure 5-1). The data can be a result of a risk analysis process, or the measurements we mentioned earlier. It's important to rely on data collected from postmortems and learnings from previous incidents that impacted customers.

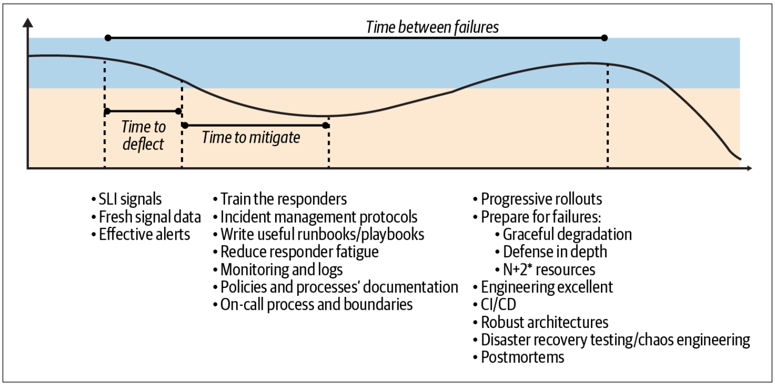


Figure 5-1. Where do you focus your efforts?

Once you have a critical mass of postmortems, you can identify patterns. It's important to let the postmortems be your guide; the investments in analyzing failure can lead you to success. For that purpose, we recommend creating a shared repository and sharing the postmortems broadly across internal teams.

Psychological Safety

It's hard to talk about postmortems without discussing psychological safety. Therefore, before diving into the details of writing postmortems, let's first talk about the psychological safety inherent in incident management culture, and discuss the value of early escalation.

If your customers are impacted, you should address the situation ASAP. This won't happen if the people in your organization do not feel safe to escalate or to scale up the size of an incident to include more people. If a company environment is such that people are discouraged from asking questions, or if there are repercussions for escalating incidents, responders may feel unsure about asking any questions at all. If that's the case, incidents are going to get a lot worse before they can get better.

Failures will happen, and you need to accept failure as normal. That's why implementing SRE principles requires a supportive and empowering culture. A key aspect of this is understanding that incidents and outages are inevitable, given the velocity of change as you constantly enhance your services with new features and add new systems. Therefore, if you don't learn from your incidents, you are missing an opportunity to improve. Consider the quote "Failure is the key to success; each mistake teaches us something" by Morihei Ueshiba, founder of Aikidoe.

As you treat operations as a software engineering problem, when things go wrong (and they will), look for the flaws in the system that allowed those things to go wrong. You want to improve things to help avoid human errors.

> Humans are never the cause of incidents, but the systems and processes that "allowed" the incidents to happen.

If an outage occurs, that's the fault of the system, not a human, because human error is a given. The goal is not to eliminate human error.

Psychological Safety When Implementing Incident Management Practices

Implementing incident management practices is an organizational change that requires some cultural prerequisites for you to innovate and learn from your mistakes. It's critical to have psychological safety and blamelessness processes in place.

> Psychological safety is a belief that one will not be punished or humiliated for speaking up with ideas, questions, concerns, or mistakes.
>
> —Dr. Amy Edmondson, Novartis Professor, Leadership and Management, Harvard Business School

Psychological safety fosters some of the main attributes of performance-oriented organizations; in particular, the treatment of failure as a learning opportunity and the acceptance of new ideas. For example, Westrum's Organizational Culture model predicts software delivery performance based on psychological safety: generative organizations are much more likely to be top performers than the other two types.[1]

Teams with higher psychological safety are more likely to harness the power of diverse ideas from teammates, beat their sales targets by 17% (compared to a 19% miss for unsafe teams), and are rated effective twice as often by executives.[2]

[1] See DevOps Culture: Westrum Organizational Culture.

[2] Google's Project Aristotle.

Psychological Safety When Handling Incidents

In risk management, it's crucial for people to know they can voice their opinions and identify problems without being penalized or ridiculed. When an incident occurs, it must be reported and it must be declared an incident. During the incident, you might have to share information about previous incidents if doing so might shed light on past mistakes (this relates to blamelessness). You also may have to hand off the incident to the next on-call engineer and suggest improvements to internal processes, tools, and features.

Without psychological safety and blamelessness, people avoid asking the right questions that may lead to identifying the root cause of an incident. As a result, teams can't learn or innovate, because they are consumed by managing impressions and fearing the consequences.

To foster psychological safety and blamelessness in your team, focus on the learning opportunity: frame each incident as something everyone can learn from, and encourage diverse ideas by inviting everyone (especially the ones who do not agree) to voice their opinions and ideas. As a leader, you should also acknowledge your own fallibility and model curiosity by asking questions.

Not casting blame

Blamelessness and psychological safety go hand in hand, and one may be a natural result of the other. Let's say there is an outage. If the first question the manager asks is "Who caused it?" it can create a culture of finger-pointing and make the team fearful of taking risks; this will prevent innovation and improvements. Instead, you should promote blamelessness:

> Blamelessness is the notion of switching responsibility from people to systems and processes.[3]

A culture of blame hinders people's ability to quickly resolve incidents and learn from mistakes, because they may want to hide information and avoid declaring incidents for fear of being punished. A blameless culture, on the other hand, allows you to focus on improvement. You want to assume that individuals act in good faith and make decisions based on the best information available. Investigating the source of misleading information is much more beneficial

[3] See more at Coursera's "Developing a Google SRE Culture" course.

to the organization than assigning blame. Therefore, support the team's design and maintenance decisions to encourage innovation and learning, and when things go wrong, focus on the systems and processes, not the people.

Learning from mistakes

Mistakes are valuable opportunities to learn and improve, but only if the correct procedural, systematic causes of the mistake are properly identified. At Google, for example, Ben Treynor Sloss sends out quarterly engineering reports of "Google's Greatest Hits and Misses" to foster an empowering culture in which we can learn from our mistakes.[4]

Additional Tips for Fostering a Psychologically Safe Environment

Incident responders need to have a certain amount of confidence to be effective responders. Even though they may be in stressful situations, it's imperative that responders feel psychologically safe while handling incidents.

This psychological safety extends to many levels:

From teammates

- Responders should not feel like they are being judged by their peers for their actions, especially when they make a mistake.
- Saying "I need help" should be rewarded, not questioned or reprimanded.

From partner teams

- Some teams may feel that *members of team X have a bad reputation for being condescending, so let's not talk to them.* Even worse, some teams embrace that culture, or they use it to avoid interacting with other teams. This attitude should not be tolerated—it creates additional tension and only slows down incident response.

[4] To learn more about learning from mistakes, see Chapter 15, "Postmortem Culture: Learning from Failure," in *Site Reliability Engineering*.

From management

- Managers are responsible for the team's psychological safety. During an incident, a manager is often not doing technical work. Instead, they are focusing on ensuring the well-being of the team—looking out for signs of stress/burnout, maybe ordering pizza while the rest of the team works on an incident. It might be as simple as a manager telling an incident responder, "Take a five-minute break to clear your head."
- Managers can also be instrumental in getting additional help from other parts of the organization.
- Managers provide the team the buffer they need from the rest of the organization, and they step in to resolve conflicts should they arise.

From the organization

- Psychological safety can thrive only if the organization embraces it in its culture. There should be a blame-free culture[5] in which the focus is on fixing the processes that led to an incident.
- The industry is filled with policies such as the *three strikes policy*, which calls for termination or a severe reprimand of individuals involved in three mistakes that affect production. Although the intent of such a policy is to encourage responders to be extra careful during an incident, it results in degraded responses ("I don't want to be the one who makes a bad call"), blame-shifting ("We didn't break it, *another* team broke it"), or hiding valuable information ("Let's not divulge the fact that we already knew about this issue").
- If leaders want their teams—and, by extension, their organizations—to thrive, they must foster a culture of respect, trust, and collaboration. This *has* to come from the top of the organization.

As noted earlier, one clear benefit of a psychologically safe environment is a reduction in escalation times. If an organization embraces a culture of collaboration, incident responders will be more likely to solicit additional help—whether from their own team or from another team in the company.

[5] See Chapter 15, "Postmortem Culture: Learning from Failure," in *Site Reliability Engineering*.

One recurring theme while reviewing incidents has always been "If only we had escalated earlier, we could have saved $XXX in lost revenue," even in teams/organizations with healthy, psychologically safe environments. It's difficult for an incident responder to ask for help—lest it be seen as a sign of weakness or unpreparedness. We've been trained to hide insecurities (even perceived ones), and taught generally to *be a hero* and *give 110% to the team*. These behaviors are actually liabilities, particularly during incident response—an overwhelmed or tired responder is more likely to make mistakes. Escalations should therefore be cheap and quick, and should come with no strings attached. Always assume the best of intentions. If it turns out that the escalation was unnecessary, find out *why* the escalation happened—maybe there was poor/missing documentation—and fix the faulty process.

Incident responders should be hyperaware of the tendency to try to do everything themselves, and instead should escalate early and often. Within one Google incident response team there is an adage: "We tell other teams that we don't mind getting paged too often—and we still don't get paged enough."

Writing Postmortems

Now that we've covered psychological safety in depth, let's move on to writing postmortems. When things break, it's your opportunity to learn from them and improve for the future. While a "terrible engineer" might think "Let's hope no one saw that," a better engineer notices that something broke and thinks "Cool! Let's tell everyone!" This is where writing postmortems comes in.

Writing a postmortem is a form of systems analysis: it's the process of diving into one of those faults that caused the incident and identifying areas to improve engineering and engineering processes. Postmortem writing isn't just an extraneous practice, but rather a core way to practice systems engineering on services, in order to drive improvements.

When writing postmortems, it's important to create a blame-free culture and processes that assume incidents *will* occur. As mentioned before, it's important to prevent failure, but realize that day-to-day failures are inevitable, especially when talking about scale. Incidents provide you and your team the opportunity to learn from them, together. Postmortems are your systematic solution to make sure you collectively learn from your mistakes, and help share that knowledge and learn from the mistakes of others as well—for example, by reading the postmortems of others.

Postmortems provide a formalized process of learning from incidents, as well as a mechanism to prevent and reduce incidents, their impact, and their complexity. For example, you may learn to avoid patches as a permanent solution. Postmortems highlight trends and prioritize your efforts. They should be blameless—this prevents side conversations ruminating about the issue, who did what, and who might be at fault. Instead of assigning blame, postmortems focus on what was learned from the incident and improvements for the future.

There is some information that every postmortem should include. For example, good postmortems include clear actions items (AIs), and the owners of and deadlines for those AIs. Remember, this is not to place blame, but to increase ownership, remove ambiguity, and make sure actions are followed up on. In addition, it's important to have a clear timeline that includes the start time of an outage, when the issue was detected, the escalation time (if applicable), the mitigation time (if applicable), the impact, and the end time of the outage. If escalation occurred, specify why and how it occurred. To avoid confusion, clarify the terminology for *incident* and *outage*, and for *incident started* and *incident detected*. We recommend keeping a "live document" during the incident as a working record of debugging and mitigation which can be used later for the postmortem. The document assists with capturing the timeline correctly and verifies that you don't miss vital AIs.

Avoid blameful language in postmortems, and practice psychological safety. Lead by example and ask lots of questions, but never seek to blame. It's about understanding the reality of the event, the actions taken, and what can be done to prevent recurrence in the future.

A Google best practice is to share the postmortem with the largest possible audience that would benefit from the lessons imparted. Transparent sharing allows others to find the postmortem and learn from it.[6]

[6] For more information, see Appendix D, "Example Postmortem," in *Site Reliability Engineering*, and public communication on a Google Compute Engine incident.

We've found that establishing a culture of blameless postmortems results in more reliable systems and is critical to creating and maintaining a successful SRE organization.

Systems Analysis for Organizational Improvement

We've talked about blameless postmortems and touched on postmortems being a form of systems analysis. However, are you really digging into your system to fully understand what happened and why? Events should be *analyzed* in order to draw conclusions, not simply recounted. The depth of analysis after an incident, or within a postmortem, asks whether events and system facets are analyzed in order to expose and explain conclusions. This is important because it increases the probability that your team will work on the right fixes after an incident.

When writing a postmortem, you should aim for the most complete and accurate picture of your system, such that fixes made are the right ones. In Figure 5-2, the circle labeled "What you THINK the problem is" reflects your understanding of your system during the incident—this is the part that is in your control. The circle labeled "What the problem actually is" reflects the actual state of your system during the incident. With a complex, technical ecosystem of many moving parts, interacting with all of the human processes of managing an incident, it's extremely difficult to truly understand all of that nuance (in fact, we once had a senior engineer try to do this, and they spent a full month trying to understand what was a 20-minute incident!). However, the deeper the analysis you conduct after an incident, the bigger the overlap in circles, and the closer you get to understanding the underlying issues (Figure 5-3).

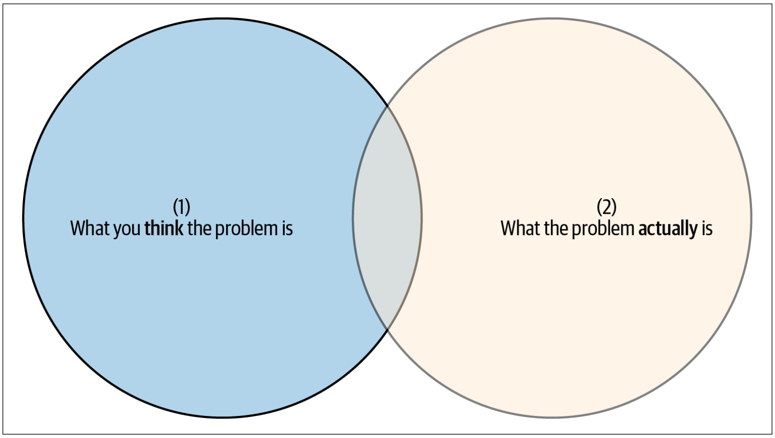


Figure 5-2. Venn diagram visualizing the gap between understanding and truth

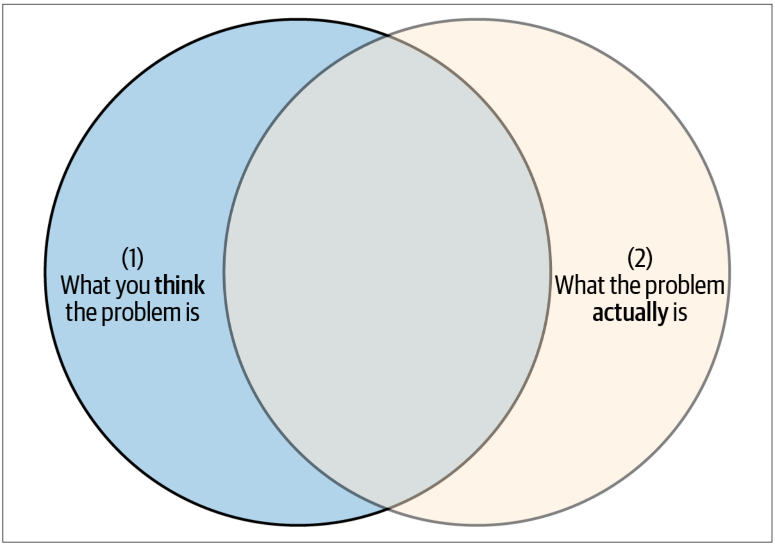


Figure 5-3. Venn diagram visualizing the benefit of systems analysis

If the incident has been mitigated and the system is stable again, does it matter that you understand the real problems? Yes. It matters because of *actionability*—or what you have the power to fix or change after an incident. These post-incident incremental system improvements help to build resilience over time. This is the third, important circle, representing the things in a system over which you have control and for which you can implement fixes (Figure 5-4).

This circle can't move either, because there will always be something out of your control which can affect the health of your systems (the weather, the size of the Earth, the speed of light).

That small intersection in the center (in set theory, notated as 1 ∩ 2 ∩ 3) is the best work that your team can do after an incident. The overlap of "What you THINK the problem is" and "What you can fix" [(1 ∩ 3) – 2] is dangerous: these are solutions that you think will help in the long term but actually won't address the real issues. You might be addressing something adjacent to the major problems, or you might be addressing a manifested symptom of another hidden issue. Assuming that you've solved an issue that hasn't really been solved is a hazardous position—made even worse by a lack of awareness of that mounting risk. If a particular incident happens again, you're left with diminished customer trust and time lost that could have been used more effectively.

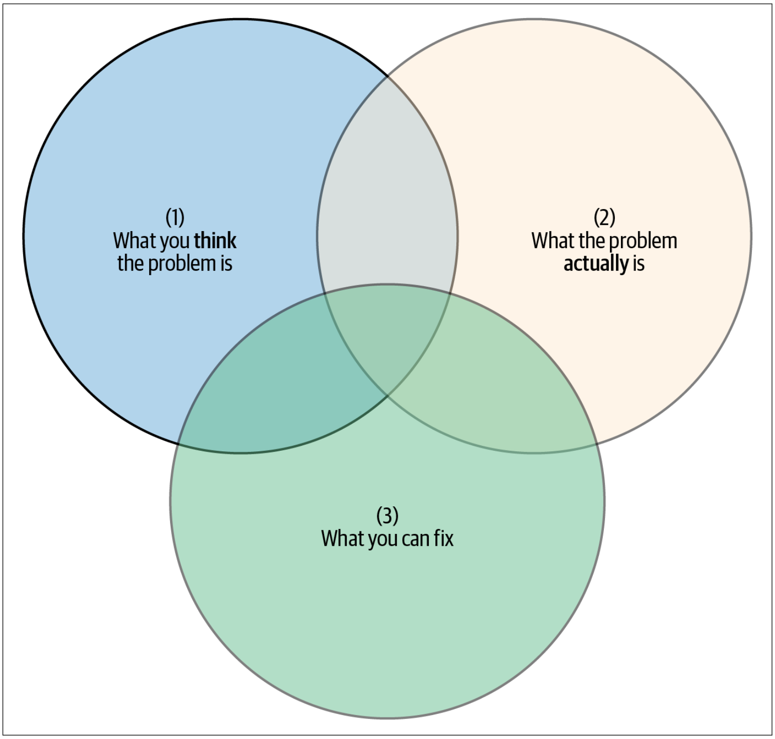

(1)
What you **think** the problem is
(2)
What the problem **actually** is
(3)
What you can fix

Figure 5-4. Venn diagram visualizing how engineering work interacts with systems analysis

As a result of a deeper systems analysis, that small piece in the center (1 ∩ 2 ∩ 3) is maximized in size, given the other two unmoving circles (Figure 5-5). In other words, you're maximizing the likelihood that the fixes you will prioritize will be effective. If you want to make sure you're targeting the right work, moving the circle is worthwhile. The key is to invest enough in systems analysis that you and your team can achieve a high probability of selecting the best possible engineering projects to improve system resilience. But there is a point of diminishing returns—for example, spending a month investigating every outage isn't a prudent use of resources. In the following section, we suggest a couple of key things that might be helpful to think about to help move the circle.

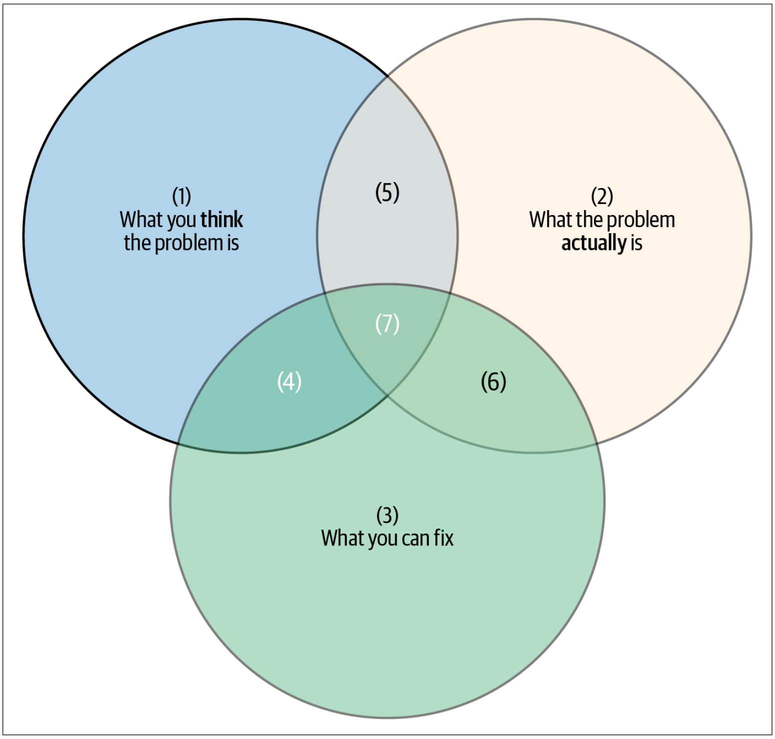

(1)
What you **think** the problem is
(5)
(2)
What the problem **actually** is
(7)
(4)
(6)
(3)
What you can fix

Figure 5-5. Venn diagram highlighting the interaction between engineering work and systems analysis

Root Cause Versus Trigger

Let's start with two key terms, *root cause* and *trigger*:

Root cause(s)

The system hazards, or how the system was vulnerable. A hazard can exist in a system for an indefinite period of time—the system environment needs to shift somehow to turn that hazard into an outage. Let's be clear: there is rarely just one root cause for an incident in a complex system. Skilled practitioners consider the underlying cause of an incident to be a web of interacting causal factors that result in a hazard state.

Trigger(s)

The circumstances that allowed the root cause(s) to turn into an incident. It's a related, but separate, thing! In order to prevent recurrences of that outage, sometimes it's important to address the root cause. Sometimes it's more reasonable to build prevention around those triggers.

The root cause and the trigger work together to create an incident. Of course, this is a bit of a simplification. Borrowing a term from medicine, the root cause contraindicates with the trigger condition to generate the resultant scenario (the incident). There isn't a one-to-one mapping of which root causes and triggers cause which types of incidents—complexity makes a whole range of outcomes possible. Let's check out some examples:

House fire

- Root cause: A gas leak
- Trigger: A sparking electrical plug near the leaky stove which ignited the leaking gas and caused the house to ignite
- Incident: A house fire (but the root cause here could have caused other incidents)

Ant infestation

- Root cause: A warm season that was comfortable for bugs and pests to thrive in the natural environment
- Trigger: Sloppy eating habits that leave lots of crumbs around
- Incident: An ant infestation

Out of Memory (OOM)

- Root cause: A config file change that introduces a memory leak
- Trigger: A surprisingly high volume of requests
- Incident: OOM

In the third (OOM) scenario, the root cause might have been put in place years before the trigger conditions existed—which is one of the fun ways in which technical debt ends up being more expensive than expected. And this root cause might not even be a bug—it can be anything that imposes a constraint on the behavior of your system. Constraints aren't inherently hazardous until the system faces an environmental condition that turns them into hazards. Let's also clarify that the trigger might not exist in a binary condition. It's not that the trigger *is* or *isn't*—the trigger condition might exist within a dynamic range, only becoming an incident when the environmental conditions of a system and the root cause interact. These two things can be seen as critical components that create the incident lifecycle.

The root cause section of the postmortem should detail some of the root causes at play and the trigger for the incident at hand. To prevent recurrences of an outage, sometimes it's important to address the root causes. Sometimes it's more reasonable to build prevention around the triggers.

However, just separating the discussion of root causes and triggers won't inherently boost the quality of your team's postmortems. All sections having an appropriate amount of content is a minimum need, but it's also important that the postmortem includes in-depth analysis, is understandable by engineers outside your team, and is actionable. Is this a recurring issue? Are mitigation steps noted, or do you need to play bug sleuth? Does the postmortem suitably explain or quantify the system's regular functioning to show the scaled comparison and impact of the failure? Seriously, if you say 89% of the product's user base was affected… what does that mean?

Isolated System Versus Holistic Stack

It's unlikely that the system affected by an incident exists in a vacuum (unless you're joining us from Hoover, Dyson, or Roomba). Unfortunately, it's a common antipattern to limit the scope of systems analysis to what appears to be broken—without considering the system context (the parts of the system's environment that are relevant to the system's functioning). Here are some things to think about that will broaden the depth of your systems analysis:

- (If applicable) is this incident reviewed as a single event, or are affiliated/correlated/child events discussed?

- Did you or any major internal customers learn about previously unknown dependencies?
- How well did end-to-end communication happen?

While the incident might just have occurred within one subsection of your overall stack, it doesn't mean your incident happened in isolation. Looking at if and how the incident affected the overall stack and members of the company may uncover insights into how things break. This can include whether your incident caused other incidents or cascading failures, or whether your company as a whole was able to communicate effectively during the incident.

Point-in-Time Versus Trajectory

In research, the technique of meta-analysis is the concatenation of multiple studies into bigger-picture conclusions. If you consider each postmortem to be a research study that conveys a point-in-time view of your system, then considering these works as a whole can help identify emerging patterns and insights. We recommend taking each postmortem as an opportunity to check in on the behavior of your system over time. Consider the following:

- Is this incident reviewed in terms of the system's trajectory over time?
- Is the same failure type recurring?
- Are there any reinforcing or balancing loops happening long term?

Part of holistic systems thinking is considering your system over time. In general, it's a good thing to never have the same incident happen twice.

We've looked at systems analysis for organizational improvement, and how it can benefit you and your team. Now let's look at a real-world example of this.

CHAPTER 6

The Mayan Apocalypse: A Real-World Example

To see some of the principles we talked about in action, we're going to dig into a real-world example of a major Google outage. We'll go through what happened, see the scaled organizational structure in action, and show how it was resolved and how we've worked to learn from this incident.

For Google, the Mayan Apocalypse was not some New Age phenomenon that led to failure during the year 2012. Rather, the Mayan Apocalypse happened June 2, 2019, with a network automation tool named Maya. Maya does flag management and organizes traffic direction over one of our networking backbones, and a tiny, tiny code shift led to an entity type being consistently mis-flagged.

Around noon, we were proceeding with planned maintenance. We finalized a list of operations and configuration changes (including on Maya) to be run over a set number of servers. When this mis-flag came into conflict with our job scheduling logic, we "discovered" a new failure mode in which jobs associated with traffic direction were de-scheduled en masse. The network traffic to/from those regions then tried to fit the de-scheduled jobs into the remaining network capacity where traffic direction was still functional, but it did not succeed. The network became congested, and our systems correctly triaged the traffic overload and auto-drained larger, less latency-sensitive traffic in order to preserve smaller, latency-sensitive traffic flows.

The traffic jam had begun. As a result, monitoring kicked off the first step in our incident management process: alerting. When a component responder receives an alert from their monitoring system, it's reflective of a change that has happened within the system they are covering. Our monitoring systems noted that our error thresholds were crossed, and they sent an automated notification to the person on call for that networking component, who began to assess what was going on.

Meanwhile, the reduced network capacity in affected regions caused spillover, and that network congestion led to cascading failures throughout our network and computing infrastructure. In general, our network prioritizes our users above our internal traffic, including employees. This is actually fine because we'd rather redirect capacity from the 99.9% of the workforce who can't help solve the problem, and use it to do the best we can for our users. The 0.1% of employees who participate in incident response usually know how to proceed and sidestep this throttle. But one of the effects of this cascading failure was significant outages of our internal tooling, which disrupted a lot of alerting and led to a huge number of pagers going off. As every on-caller switched into incident response mode, they noticed service unavailability due to networking issues. The network component on call quickly identified the cause of the network congestion, but the same network congestion which was creating service degradation also slowed their ability to restore the correct configurations.

Everyone wanted to best support their users and understand the anticipated trajectory of service restoration, so the original network component on-caller suddenly had a lot of company.

We have three classifications of components at Google:

- Infrastructure components such as a networking pipeline, or storage service.
- Product-service components such as YouTube streaming, or the frontend of Google Search.
- Internal service components like monitoring, zero trust remote access, and Maya and fleet management. And everything in this bucket was having a bad time.

The widespread dependency meant that no one could move forward until the network component on call resolved the issue. Other on-callers began chiming in, offering assistance, and asking questions about when *their* service would be alive again. The intended parallelism from having so many distinct responders was not providing accelerated mitigation. Root causes and second-order effects began to blur; one team's cause was another team's effect, and everyone was trying to contribute their knowledge. While everyone is expert in *their* system's stack, most didn't have a full-system overview of which tooling paths were rendered unusable.

Why? Paths which never touched the congested network were fine. Paths which did touch the congested network were OK if, *at that point*, the path looked like an external user, because we'd assigned them priority. So, services we offer to external users were available—things like video calls or editing documents. However, if the path was something fundamentally internal, such as job or flag control or Maya configurations, it was deprioritized and stuck.

We were watching a volcano erupt and then, 20 minutes later, coming to the conclusion that our issue "might be lava related."

One hour into the outage, one component responder noted that the system-of-systems issues impacting our infrastructure were too pervasive, and coordinated communications surrounding the incident were turning into chaos and discord. At this point, more than 40 teammates had joined the incident response communication channel, chiming in to try to help. Measuring the impact showed us half the globe. Google Cloud, Gmail, Google Calendar, Google Play, and other services were affected—preventing businesses from operating, preventing employees from being productive, and preventing people from communicating with one another. Some employees were trying to use the fragmented services that didn't need the damaged network, while others had given up.

With nearly 40 people involved, there wasn't enough mental space for our networking hero to work out the appropriate mitigations, coordinate implementing those mitigations, communicate widely to all stakeholders, and manage expectations. So, they escalated. Our network component on-caller paged Tech IRT; their page reached a number of Tech IRT members within an appropriate time zone, and those available to work on the incident signaled their availability. Because the incident was so far reaching, many were already involved in the incident. Several of our Tech IRT responders didn't take the role of incident commander, because they were members or managers of teams that worked on networking and could help address the primary root cause, so they chose to assist with operations instead.

The member of Tech IRT who accepted the role of incident commander hadn't previously worked on the networking components that were affected by the failure, but they were able to assess the state of our SoS and assess the state of the people responding to the outage. Using their training, this person accessed our production systems using a mechanism that immediately identifies their actions as "incident response," and was able to subvert the "degraded internal traffic" flagging. Once there was a little headroom for internal traffic, they directed the networking on-caller to jump in and make things happen.

While that was ongoing, they quickly imposed structure and organization on the communications that were occurring and on everyone who was trying to "pitch in." Once this frenzy of chaotic engineering energy was organized, everyone started making progress—together. They could more clearly track the ongoing status of different systems, and see the pace with which mitigative actions were being rolled out. With this administrivia no longer burdening our network component responder, they—and their team, who had shown up to help—had the space to implement an appropriate mitigation plan. This included shedding a lot of load in order to buy system headroom for a healthy reboot and some emergency-forced config changes.

Once the path forward to mitigate the incident was in motion, the Tech IRT member focused on driving the incident toward closure. They set some exit criteria for when we could close the incident, made sure that other systems were supported in any recovery actions they needed to perform, and then made sure the involved response team could hand off and detach.

After the incident concluded and normal service was restored, an in-depth postmortem took place to analyze the incident and understand the nuances of its root causes and the emergent properties that were brought to light through these failure modes. The networking teams involved have since worked on some really cool initiatives to restructure Maya and prevent this failure mode, and similarly-possible-but-previously-not-considered failure modes from ever plaguing our systems again.

Finally, we rewarded the folks involved with internal profile badges, honorific memes, and bonuses. A really severe incident is, for most, the worst day of their career. Offering something small provides a subtle incentive for everyone to contribute to the postmortem, help us learn, and continually grow more resilient.

CHAPTER 7

Conclusion and Moving Forward

We looked at the basics of incidents, and took a look at the incident management lifecycle: preparedness, response, and recovery. It's a lot to process, but you might now be wondering, "What's next?"

Your first call to action is this: learn to use incident management only when it's appropriate. Incident response is a human-expensive activity. A person, often several people, needs to be involved throughout the drive from initial alerting to resolution. The act of incident response is intended to put in place mitigations that correct problems while they are happening, in order to buy time to make decisions about priorities. This means that regular product fixes might not be rolled out and long-term plans and improvements might not be prioritized. Incident response might mean that SLOs are violated or customer commitments can't be met. It also means that the employees working on incident response are going to feel it.

It's well documented that first responders to physical incidents are at heightened risk of burnout and responder fatigue; this same trend also applies to individuals who work on nonphysical incidents—namely, anyone whose job can involve work–life imbalance, extremes of activity, or a possible lack of control. These are common factors in technical incident management jobs, which means that employees can feel the effects and career consequences of burnout. The risks here involve, at best, low performance, and at worst, employee attrition. Because of this possibility of burnout-related employee effects, it becomes imperative for a company to do incident management as well as possible, and as little as possible.

Your next action item is to treat incident management as a critical operational discipline at which you want to be good. So, what does it mean to "be good" at incident management? It means your team (not just any one individual) actively works to improve all parts of this cycle. Although this is dull in comparison to the dramatic mental picture of a few superhero firefighters swooping in to save the day, a heroism mindset is harmful. The less-exciting work of slowly and carefully improving incident preparedness; developing the tools, techniques, and communication pipelines to respond to incidents well; and then prioritizing sustainable and scalable engineering are what comprise a strong incident management practice. By seeing everything in a continuous and interconnected loop, everyone is important and you avoid placing the blame on any one person or system component. The practice of blamelessness fosters the kind of psychologically safe workplace where your people can thrive and build great products. These are the types of things that helped Google get through a period of tremendous uncertainty in recent global history, and these are the types of things that can help improve resilience at your company too.

In general, don't throw incident management at every potential problem or type of problem. Use incident management sparingly and appropriately in order to avoid burning out your team members. Stop managing the incident when you're done managing the incident, and start doing the engineering work needed to fix your longer-running issues or risks. Identify and use some other tools that might be at your disposal.

Additional Reading

- Monitoring from *The Site Reliability Workbook*
- Incident Response from *The Site Reliability Workbook*
- Postmortem Culture: Learning from Failure from *The Site Reliability Workbook*
- Postmortem Action Items: Plan the Work and Work the Plan
- "Shrinking the Impact of Production Incidents Using SRE Principles—CRE Life Lessons"
- ""Shrinking the Time to Mitigate Production Incidents—CRE Life Lessons"

Bibliography

"Google Data Center FAQ". Data Center Knowledge, 19 March 2017.

Aleksandra. "63 Fascinating Google Search Statistics". SEOtribunal, 26 September 2018.

"Incident Command System Resources". FEMA, The US Department of Homeland Security, 26 June 2018.

Beyer, Betsy, Chris Jones, Niall Richard Murphy, and Jennifer Petoff, eds. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, 2016.

"Data Access and Restrictions". Google Workspace Security Whitepaper, October 2021.

Treynor Sloss, Benjamin. "An Update on Sunday's Service Disruption". Inside Google Cloud (blog), Google Cloud, 3 June 2019.

Acknowledgments

The authors thank Jennifer Mace, Hazael Sanchez, Alexander Perry, Cindy Quach, and Myk Taylor for their contributions to this report.

About the Authors

Ayelet Sachto is a site reliability engineer in GKE SRE, formerly strategic cloud engineer and leading PSO-SRE efforts in EMEA @Google UK. Throughout her 17-year career, she developed and designed large-scale applications and data flows while implementing DevOps and SRE's methodologies. She is the author of numerous technical articles, talks, and trainings, including "SRE Fundamentals in 3 Weeks" (an O'Reilly course), and has spoken at dozens of conferences and led hundreds of workshops. Ayelet is also an active member in the tech community and a mentor. In her spare time, Ayelet loves creating things, whether it's a dish in the kitchen, a piece of code, or impactful content.

Adrienne Walcer is a technical program manager in SRE at Google. She is focused on resilience: reducing the impact of large scale incidents on Google services, infrastructure, and operations. A previous contributor to Google's O'Reilly publications (*A Practical Guide to Cloud Migration*), Adrienne was also a featured speaker on scaled incident management at the final USENIX LISA conference (LISA21). Before Google, Adrienne was a data scientist at IBM Watson Health (formerly, Explorys Inc.), and worked in biostatistics at Strong Memorial Hospital and the Cleveland Clinic. She holds a masters degree in systems engineering from George Washington University and a bachelor's degree from the University of Rochester. In her free time, Adrienne enjoys Dungeons & Dragons and volunteers with Second Harvest Food Bank.