Debugging Incidents in Google's Distributed Systems

HOW EXPERTS DEBUG PRODUCTION ISSUES IN COMPLEX DISTRIBUTED SYSTEMS

CHARISMA CHAN AND BETH COOPER

Google has published two books about SRE (Site Reliability Engineering) principles, best practices, and practical applications.[1,2] In the heat of the moment when handling a production incident, however, a team's actual response and debugging approaches often differ from ideal best practices.

This article covers the outcomes of research performed in 2019 on how engineers at Google debug production issues, including the types of tools, high-level strategies, and low-level tasks that engineers use in varying combinations to debug effectively. It examines the research approach used to capture data, summarizing the common engineering journeys for production investigations and sharing examples of how experts debug complex distributed systems. Finally, the article extends the Google specifics of this research to provide

some practical strategies that you can apply in your organization.

RESEARCH APPROACH

As this study began, its focus was to develop an empirical understanding of the debugging process, with the overarching goal of creating optimal product solutions that meet the needs of Google engineers. We wanted to capture the data that engineers need when debugging, when they need it, the communication process among the teams involved, and the types of mitigations that are successful. The hypothesis was that commonalities exist across the types of questions that engineers are trying to answer while debugging production incidents, as well as the mitigation strategies they apply.

To this end, we analyzed postmortem results over the last year and extracted time to mitigation, root causes, and correlated mitigations for each. We then selected 20 recent incidents for qualitative user studies. This approach allowed us to understand and evaluate the processes and practices of engineers in a real-world setting and to deep-dive into user behavior and patterns that couldn't be extracted by analyzing trends in postmortem documents.

The first step was trying to understand user behavior: At the highest level, what did the end-to-end debugging experience look like at Google? The study was broken down into the following phases (which are unpacked in the sections that follow):

- *Phase 0* – Define a way to segment the *incident responder* and *incident type* populations.
- *Phase 1* – Audit the postmortem documentation from

a spread of actual Google incidents.

➡ *Phase 2* – Conduct in-depth user interviews with first responders who worked on those incidents.

➡ *Phase 3* – Map the responders' journeys across those incidents, detailing common patterns, questions, and steps taken.

Phase 0: Segment incident responder and incident type populations

The preliminary approach to segment the population under study was designed to ensure that a sufficiently broad set of incidents and interviewees was included, from which we could capture a comprehensive set of data.

Incident Responders

First, the *incident responder* (or *on-callers*) were segmented into two distinct groups: SWEs (software engineers), who typically work with a product team, and SREs (Site Reliability Engineers), who are often responsible for the reliability of many products. These two groups were further segmented according to tenure at Google. We found the following behaviors across the different user cohorts:

SWE vs. SRE mental models and tools

SWEs are more likely to consult logs earlier in their debugging workflow, where they look for errors that could indicate where a failure occurred.

SREs rely on a more generic approach to debugging: Because SREs are often on call for multiple services, they apply a general approach to debugging based on known characteristics of their system(s). They look for common

failure patterns across service health metrics (e.g., errors and latency for requests) to isolate where the issue is happening, and often dig into logs only if they're still uncertain about the best mitigation strategy.

Experience level of the incident responder

Newer engineers are more likely to use recently developed tools, while engineers with extensive experience (10-plus years running complex, distributed systems at Google) tend to use more legacy tools. Intuitively, this finding makes sense—people tend to use the tools they are most comfortable with, particularly in emergency situations.

Incident Types

We also examined incidents across the following dimensions, and found some common patterns for each:

- *Scale and complexity.* The larger the blast radius (i.e., its location(s), the affected systems, the importance of the user journey affected, etc.) of the problem, the more complex the issue.
- *Size of the responding team.* As more people are involved in an investigation, communication channels among teams grow, and tighter collaboration and handoffs between teams become even more critical.
- *Underlying cause.* On-callers are likely to respond to symptoms that map to six common underlying issues: capacity problems; code changes; configuration changes; dependency issues (a system/service my system/service depends on is broken); underlying infrastructure issues (network or servers are down); and external traffic issues. Our investigation intentionally did not look at security or

data-correctness issues outside the scope of the tools focused on in this work.

➡ *Detection.* On-callers learn about issues through human or machine detection that is based on availability or performance problems. Some common mechanisms include alerts on the following: white-box metrics; synthetic traffic; SLO (service-level objective) violations; and user-detected issues.

Phase 1: Postmortem documentation analysis

Once the different categories of incidents were determined, we read the postmortems for the 20 incidents identified for qualitative studies, mapping the steps responders took for each case. This approach allowed us to validate the common factors that affect how responders handled these incidents and the challenges they faced. We could also ensure that the incidents selected for deep-dive analysis were distributed across the dimensions, as just described.

Google has a strong culture of blameless postmortems.[4] It is common for teams to look at the history of their failures to ensure that their services are continuing to run reliably. Because of this, postmortem documents are readily available internally and were an invaluable resource for analyzing debugging behavior. Detailed chat transcripts linked to these postmortems helped form a base understanding of what happened, when it happened, and what went wrong. We could then start mapping a prototype of the debugging journey. Future research could extend this work by applying natural-language processing to further validate response patterns in the incident response chats.

Phase 2: In-depth interviews

To round out this study, in-depth interviews were conducted with the first responders identified in these 20 postmortems so any gaps in the postmortem document could be filled in. These data sources added significant color to the debugging journey we were mapping, and surfaced a core set of building blocks that make up the overall debugging process.

Building blocks are often repeated as the user investigates the issue, and each block can happen in a nonsequential and, sometimes, cyclical order.

Phase 3: Mapping the responders' journeys

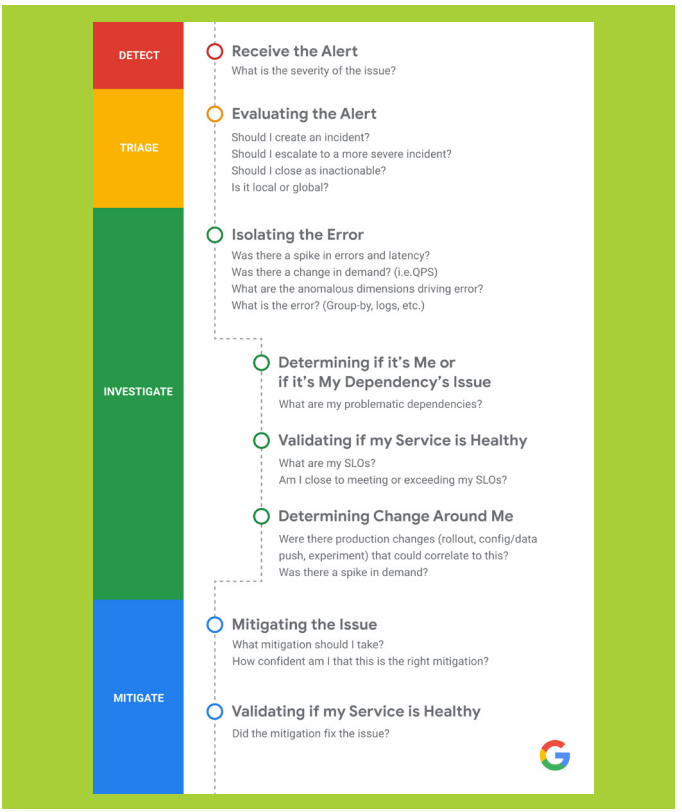
This study allowed us to generate snapshots of what an actual incident investigation lifecycle looks like at Google. By mapping out each responder's journey and then aggregating those views, we extracted common patterns, tools, and questions asked around debugging that apply to virtually every type of incident. Figure 1 is a sample of the visual mapping of the steps taken by each of the responders interviewed.

COMMON PATTERNS AROUND DEBUGGING

A typical canonical debugging journey consists of the stages and sub-journeys shown in figure 2 and described in this section. *These building blocks are often repeated as the user investigates the issue, and each block can happen in a nonsequential and, sometimes, cyclical order.*

During the detection to mitigation stages, investigations are typically time sensitive—especially when the issue affects the end-user experience. An on-caller will always try to mitigate the issue or "stop the bleeding" before uncovering the root cause. After mitigation, on-callers and

FIGURE 1: BUILDING BLOCKS

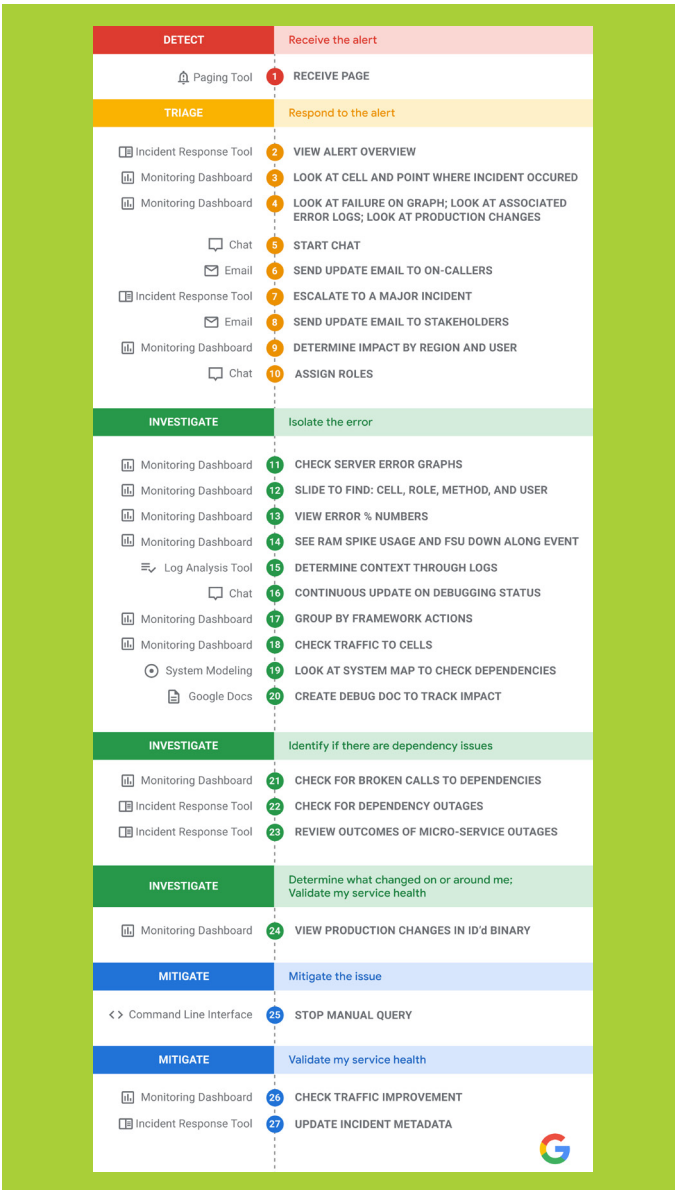


developers often perform a deeper analysis of the code and apply measures to prevent a similar situation from recurring.

Detect

The on-caller discovers the issue via an alert, a customer escalation, or a proactive investigation by an engineer on the team. A common question would be: What is the severity of this issue?

FIGURE 2: **USER JOURNEY**



Triage loop

The on-caller's goal is to assess the situation quickly by examining the situation's blast radius (the severity and impact of the issue) and determining whether there is a need to escalate (pull in other teams, inform internal and external stakeholders). This stage can happen multiple times in a single incident as more information comes in.

Common questions include: Should I escalate? Do I need to address this issue immediately, or can this wait? Is this outage local, regional, or global? If the outage is local or regional, could it become global (for example, a rollout contained by a canary analysis tool likely won't trigger a global outage, whereas a query of death triggered by a rollout that is now spreading across your systems might)?

Investigate loop

The on-caller forms hypotheses about potential issues and gathers data using a variety of monitoring tools to validate or disprove theories. The on-caller then attempts to mitigate or fix the underlying problem. This stage typically happens multiple times in a single incident as the on-caller collects data to validate or disprove any hypotheses about what caused the issue.

Common questions include: Was there a spike in errors and latency? Was there a change in demand? How unhealthy is this service? (Is this a false alarm, or are customers still experiencing issues?) What are the problematic dependencies? Were there production changes in services or dependencies?

Sometimes a mitigation attempt can make the issue worse or cause an adverse ripple effect on one of its dependent services.

Mitigate loop

The on-caller's goal is to determine what mitigation action could fix the issue. *Sometimes a mitigation attempt can make the issue worse or cause an adverse ripple effect on one of its dependent services.* Remediation (or full resolution of the issue) usually takes the longest of all the debugging steps. This step can, and often does, happen multiple times in a single incident.

Common questions include: What mitigation should be taken? How confident are you that this is the appropriate mitigation? Did this mitigation fix the issue?

Resolve/root-cause loop

The on-caller's goal is to figure out the underlying issue in order to prevent the problem from occurring again. This step typically occurs after the issue is mitigated and is no longer time sensitive, and it may involve significant code changes. Responders write the postmortem documentation during this stage.

Common questions include: What went wrong? What's the root cause of the problem? How can you make your processes and systems more resilient?

Communication

Throughout the entire process, incident responders document their findings, work with teammates on debugging, and communicate outside of their team as needed.

OBSERVABILITY DATA

In every single interview, on-callers reported that they

started working with time-series metrics that indicate the health of a given service, performing a breadth-first search to identify which components of the system were broken. The majority of the teams that were interviewed evaluated the following items:

- RPC (remote procedure call) latency and error metrics (similar to the metrics derived from the open-source gRPC libraries).
- Change in external traffic, including QPS (queries per second).
- Change in production such as rollouts, configuration pushes, and experiments.
- Underlying job metrics such as memory and CPU consumption.

Both alerts and realtime dashboards use these metrics. On-callers typically used logs and traces only after they identified a component as broken, and they then needed to drill down to the specific issue.

ANECDOTES FROM THE FRONT LINE

Some of the interviewees applied SRE best practices to debug complex distributed systems, methodically eliminating their theories on what could go wrong, applying temporary mitigations to prevent user pain, and, finally, successfully resolving and root-causing the problem that set off the outage in the first place.

Many other responders hit unexpected roadblocks. Some responders were impacted by a complex set of changes throughout the stack that occurred simultaneously. Therefore, it was extremely challenging to isolate the actual issue and figure out how to resolve

it. Other responders cited process and awareness issues: Some did not fully understand how their production tooling worked, or the appropriate standard course of action to take. Some responders wound up unintentionally applying bad changes to production.

Following are some (anonymous) stories to illustrate successful and problematic debugging sessions. These anecdotes are intended to show that even with the most experienced engineers, great technology, and powerful tooling, things can—and do—go wrong in unexpected ways.

Some responders wound up unintentionally applying bad changes to production.

An exemplary debugging journey

The following is an example of a successful debugging session, where the SRE follows best practices and mitigates a service-critical issue in less than 20 minutes.

While sitting in a meeting, the SRE on-caller receives a page informing her that the front-end server is seeing a 500 server error. While she's initially looking at service health dashboards, a pager-storm starts, and she sees many more alerts firing and errors surfacing. She responds quickly and immediately identifies that her service isn't healthy.

She then determines the severity of the issue, first asking herself how many users are impacted. After looking at a few error rate charts, she confirms that a few locations have been hit with this outage, and she suspects that it will significantly worsen if not immediately addressed. This line of questioning is referred to as the *triage loop*, similar to triage processes used in health care (for example, emergency rooms that sort patients by urgency or type of service). The SRE needs to determine

if the alert is noise, if she needs to handle it now, and whether to escalate the issue to other teams and stakeholders.

Now that she knows this is a real and relatively severe issue, the SRE starts pulling in other people from her team to help with the investigation. She also sets up communication channels to inform other teams that may be affected, and to let them know her team is addressing the outage.

She then focuses on temporarily mitigating the issue for end users. She tasks a teammate with ensuring that traffic isn't routed to any of the unhealthy locations and configuring load balancers to avoid sending traffic to impacted locations. For the moment, this action stops the issue from propagating, which leaves her free to conduct a deeper investigation using monitoring data.

Next, she asks a series of questions that help her narrow down the potential cause and figure out how best to mitigate the issue permanently. She largely uses time-series metrics (e.g., Cloud Monitoring metrics[3]) to help answer these questions quickly:

➡ *To narrow down the breadth of the investigation:* Which specific parts of the service are unhealthy? Are the errors coming from the front end or back end? Are there "slices" of data that are problematic? Are there outliers in the data?

➡ *To identify the severity of the issue and rule out causes:* Is the shape of the graph a step (something changed suddenly and remained unchanged), a spike (something changed, then stopped), or a slope (a gradual rollout is happening)? How quickly did the error rate ramp up?

➡ *To identify the severity:* What is the blast radius? (If errors occur globally, this indicates a severe issue that will most likely have end-user impact.)

➡ *To rule out underlying causes:* When did the problem start? What production events in the service or in its dependencies correlate with this issue?

Once the issue is mitigated, the SRE drills into logs and traces, confirming that a new line of code was crashing the jobs in the regions with issues. She decides to roll back to the last stable version of the service, and validates that the issue is resolved when the affected locations are brought back online.

Debugging journey where the tooling failed to support the on-caller

The following is an example of a journey where Google on-callers hit unexpected hurdles as they debugged, and where applying best practices could have reduced the time to mitigation.

The on-caller receives a page that informs him that the service's overall server-side availability SLO (service-level objective) was down from 99.9 percent to 91 percent, and that specific user actions failed. He begins his investigation by looking at graphs of metrics that confirm (1) when the error rate started to increase; (2) errors were mostly caused by timeouts; and (3) request durations were about equal to the duration of the timeout. He then slices the metrics to the failing user actions identified before, checks the associated server errors and queries-per-second metrics, and digs into server logs to find specific errors. Up to this point, he has followed common practices for debugging.

At the same time, another on-caller for a back-end service dependency notices that the service is nearing its quota limitations and suspects that this situation might have an impact on the investigation. This on-caller tries to allocate some quota through a configuration change, hoping to alleviate the problem. Because of a misunderstanding in the configuration push tooling, however, this change accidentally removes a back-end server in one location instead of adding quota, which increases the error rates in the other locations. Additionally, since he considered this change to be safe, the on-caller didn't monitor the rollout of the updated configuration as closely as best practices recommend, and initially missed indicators that overall capacity was actually reduced because of the removed location. At this point, the on-caller breaks from best practices by performing a global push of a nonvalidated configuration that includes a completely unrelated change—the action of dropping a back end should be separate from adding capacity.

While this is happening, the first on-caller goes deep in the logs and finds "permission-denied" errors increased at the time the back-end server was removed. He does this through a breadth-first search of a number of the supporting back ends and an analysis of their aggregated logs. Here, he notices that when one server was removed, more requests were funneling to the servers that were experiencing issues. Only after digging into logs and opening a number of tools is he able to connect the errors to the configuration change in the dependency.

Better tooling could have prevented the user from performing an unanticipated change. Tooling could also

have helped validate what the change would actually do. Additionally, better tooling to support monitoring the effects of the changes to the system could have helped the on-callers draw these conclusions earlier.

The on-callers then connect to share their findings. Once connected, the first on-caller rolls back the configuration push that reduced capacity, identifies the back-end dependency that changed the permission errors, and works with the back-end team to get bad changes rolled back.

TRANSLATING INSIGHTS INTO CONCRETE ACTION

If you are responsible for running a distributed service, you might find yourself dealing with scenarios similar to what the teams we interviewed experienced. Our study revealed that teams that apply the following principles are typically able to mitigate service problems faster.

Establish SLOs and accurate monitoring

You need to have SLOs and/or metrics that you can alert and optionally report on. These should accurately reflect user pain and allow for slicing by failure domains. These should also be associated with alerts that have clear next steps and links to the most important information.

Triage effectively

Once you have the prerequisites of SLOs and accurate monitoring in place, you need to be able to quickly determine both the severity of user pain and the total blast radius. You should also know how to set up the proper communication channels based on the severity of the issue.

Mitigate early

Documenting a set of mitigation strategies that are safe for your service can help on-callers temporarily fix user-facing issues and buy your team critical time to identify the root cause. For more information on implementing generic mitigations, see "Reducing the Impact of Service Outages with Generic Mitigations with Jennifer Mace."[5] The ability to easily identify what changed in your service— either in its critical dependencies or in your user traffic—is also helpful in determining what mitigation attempt to move forward with. As mentioned in the exemplary debugging case, asking a series of common questions and having metrics, logs, and traces can help speed up the process of validating your theories about what went wrong.

Apply established mitigation strategies for common issues

Although every service is different, the following patterns emerged in the underlying issues we examined and the mitigations associated with them. When you're dealing with a problem that you've never seen before, it can be helpful to think about what type of issue your service is facing, the questions you should ask, and the associated mitigations based on the answers.

➡ *Service errors.* This was the most common cause for an alert firing in our study. As such, it also had the largest variety of mitigations. Some factors to consider in determining mitigation strategies include: (1) Are the errors occurring globally? Check for correlated rollouts, configuration/data changes, and experiments. (2) Are the incoming QPS spiking? Add capacity and/or start load shedding to drop traffic that your service can't handle. (3)

Is a bad actor causing a change in QPS? If so, block the user.

➥ *Performance.* Latency can make for a bad user experience and degrade into errors over time. These issues can be difficult to debug if there is no obvious correlated capacity or production change. Typically, responders look through traces to identify which components in the stack are affected and try to determine a solution from there.

➥ *Capacity.* Capacity issues are some of the easiest to spot, especially if you have capacity-specific alerts. Like errors and performance issues, these can manifest as both fast and slow burns. If a service is going to run out of capacity immediately, teams typically ask for more capacity in an "emergency loan" to scale up their service (or they may attempt to scale out). For a slow burn, responders perform additional analyses and planning to determine if there are other underlying issues. These types of alerts surface only when automated capacity systems hit their authorized maximum, and acquiring more resources requires human intervention.

➥ *Dependency issues.* A critical dependency—even if it's deep within the service stack—can contribute to the failure of the entire service. Knowing your hard dependencies (those in the critical path of your code) and being able to view the health of these dependencies can be helpful in ruling out whether the problem actually lies with another service.

➥ *Debugging microservices.* Most of the teams we interviewed have a microservice architecture. Frequently, the error may be deeper in the stack than where it manifested to the on-caller. Similar to debugging dependencies, it's helpful to be able to traverse the stack

Related articles

The Calculus of Service Availability
You're only as available as the sum of your dependencies.
Ben Treynor, Mike Dahlin, Vivek Rau, Betsy Beyer
https://queue.acm.org/detail.cfm?id=3096459

Why SRE Documents Matter
How documentation enables SRE teams to manage new and existing services
Shylaja Nukala and Vivek Rau
https://queue.acm.org/detail.cfm?id=3283589

Weathering the Unexpected
Failures happen, and resilience drills help organizations prepare for them.
Kripa Krishnan
https://queue.acm.org/detail.cfm?id=2371516

quickly, associate production changes, and understand service architecture.

CONCLUSIONS

SREs continuously strive to improve systems and expose vulnerabilities in order to limit the probability of failures, near misses, and inefficiencies in production. Even under the most ideal conditions, things inevitably go wrong. By surfacing, preserving, and disseminating the commonalities—both positive and negative—in the debugging workflow, the aim is to prevent the same class of problem from recurring, or, when prevention isn't possible, to minimize the duration or impact of unavoidable outages. Hopefully, other organizations can apply these findings in practice too.

References

1. Beyer, B., Jones, C., Petoff, J., Murphy, N. R., eds. 2016. *Building Secure and Reliable Systems*. O'Reilly Media; https://landing.google.com/sre/books/.
2. Beyer, B., Murphy, N. R., Rensin, D. K., Kawahara, K., Thorne, S., eds. 2018. *The Site Reliability Workbook*. O'Reilly Media; https://landing.google.com/sre/books/.

3. Google Cloud. 2020. Metric list; https://cloud.google.com/monitoring/api/metrics.
4. Lunney, J., Lueder, S. 2017. *Postmortem culture: learning from failure*. O'Reilly Media; https://landing.google.com/sre/sre-book/chapters/postmortem-culture/.
5. Mace, J. 2019. Spotlight on Cloud: Reducing the Impact of Service Outages with Generic Mitigations with Jennifer Mace. O'Reilly Media; https://www.oreilly.com/library/view/spotlight-on-cloud/0636920347927/.

Beth Cooper *is a Product Manager at Google NYC. She focuses on building Google scale monitoring for both site reliability and software engineers. Prior to Google, she worked on Microsoft Azure building products for cloud and datacenter automation.*

Charisma Chan *is a user experience design researcher at Google UK in London. Prior to joining Google, she led design research and strategy for consumer and enterprise products in the financial services and media sectors and she holds a bachelor's degree from Cornell University in Ithaca, New York.*