# Effective Troubleshooting

**Written by Chris Jones**

> *Be warned that being an expert is more than understanding how a system is supposed to work. Expertise is gained by investigating why a system doesn't work.*
>
> Brian Redman

> *Ways in which things go right are special cases of the ways in which things go wrong.*
>
> John Allspaw

Troubleshooting is a critical skill for anyone who operates distributed computing systems—especially SREs—but it's often viewed as an innate skill that some people have and others don't. One reason for this assumption is that, for those who troubleshoot often, it's an ingrained process; explaining *how* to troubleshoot is difficult, much like explaining how to ride a bike. However, we believe that troubleshooting is *both* learnable and teachable.

Novices are often tripped up when troubleshooting because the exercise ideally depends upon two factors: an understanding of how to troubleshoot generically (i.e., without any particular system knowledge) and a solid knowledge of the system. While you can investigate a problem using only the generic process and derivation from first principles,[58] we usually find this approach to be less efficient and less effective than understanding how things are supposed to work. Knowledge of the system typically limits the effectiveness of an SRE new to a system; there's little substitute to learning how the system is designed and built.

Let's look at a general model of the troubleshooting process. Readers with expertise in troubleshooting may quibble with our definitions and process; if your method is effective for you, there's no reason not to stick with it.

## Theory

Formally, we can think of the troubleshooting process as an application of the hypothetico-deductive method:[59] given a set of observations about a system and a theoretical basis for understanding system behavior, we iteratively hypothesize potential causes for the failure and try to test those hypotheses.

In an idealized model such as that in Figure 12-1, we'd start with a problem report telling us that something is wrong with the system. Then we can look at the system's telemetry[60] and logs to understand its current state. This information, combined with our knowledge of how the system is built, how it should operate, and its failure modes, enables us to identify some possible causes.

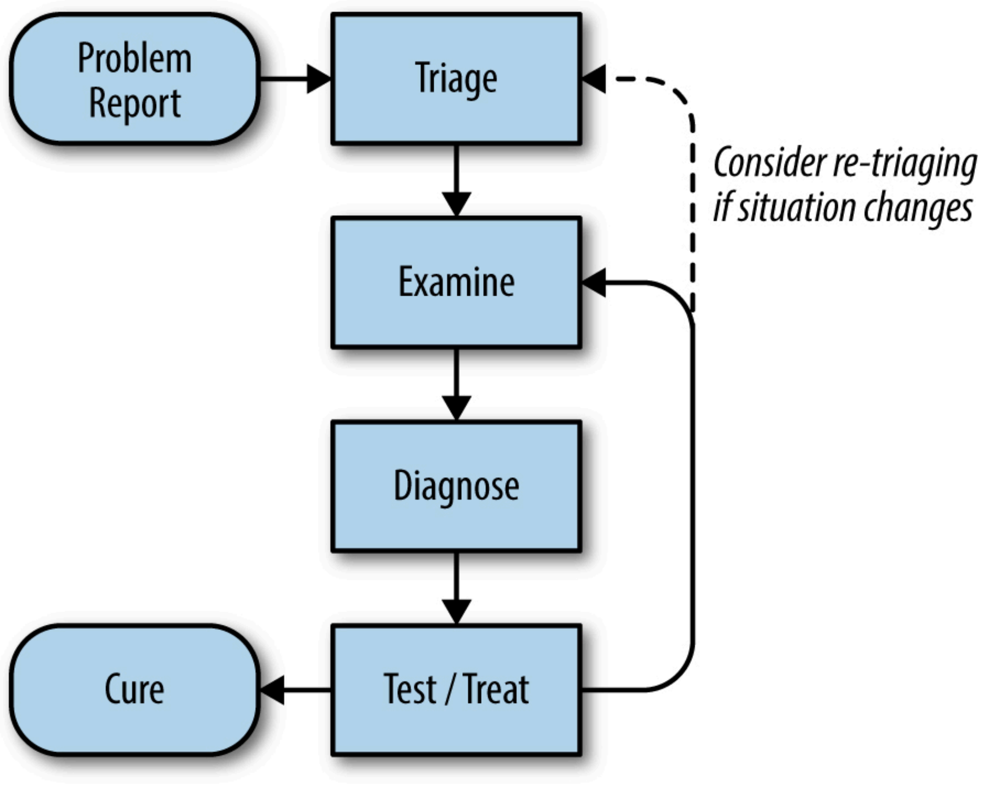

**Figure 12-1. A process for troubleshooting**

We can then test our hypotheses in one of two ways. We can compare the observed state of the system against our theories to find confirming or disconfirming evidence. Or, in some cases, we can actively “treat” the system—that is, change the system in a controlled way—and observe the results. This second approach refines our understanding of the system’s state and possible cause(s) of the reported problems. Using either of these strategies, we repeatedly test until a root cause is identified, at which point we can then take corrective action to prevent a recurrence and write a postmortem. Of course, fixing the proximate cause(s) needn’t always wait for root-causing or postmortem writing.

## Common Pitfalls

Ineffective troubleshooting sessions are plagued by problems at the Triage, Examine, and Diagnose steps, often because of a lack of deep system understanding. The following are common pitfalls to avoid:

- Looking at symptoms that aren’t relevant or misunderstanding the meaning of system metrics. Wild goose chases often result.
- Misunderstanding how to change the system, its inputs, or its environment, so as to safely and effectively test hypotheses.
- Coming up with wildly improbable theories about what’s wrong, or latching on to causes of past problems, reasoning that since it happened once, it must be happening again.
- Hunting down spurious correlations that are actually coincidences or are correlated with shared causes.

Fixing the first and second common pitfalls is a matter of learning the system in question and becoming experienced with the common patterns used in distributed systems. The third trap is a set of logical fallacies that can be avoided by remembering that not all failures are equally probable—as doctors are taught, “when you hear hoofbeats, think of horses not zebras.”[61] Also remember that, all things being equal, we should prefer simpler explanations.[62]

Finally, we should remember that correlation is not causation:[63] some correlated events, say packet loss within a cluster and failed hard drives in the cluster, share common causes—in this case, a power outage, though network failure clearly doesn't cause the hard drive failures nor vice versa. Even worse, as systems grow in size and complexity and as more metrics are monitored, it's inevitable that there will be events that happen to correlate well with other events, purely by coincidence.[64]

Understanding failures in our reasoning process is the first step to avoiding them and becoming more effective in solving problems. A methodical approach to knowing what we do know, what we don't know, and what we need to know, makes it simpler and more straightforward to figure out what's gone wrong and how to fix it.

## In Practice

In practice, of course, troubleshooting is never as clean as our idealized model suggests it should be. There are some steps that can make the process less painful and more productive for both those experiencing system problems and those responding to them.

### Problem Report

Every problem starts with a problem report, which might be an automated alert or one of your colleagues saying, "The system is slow." An effective report should tell you the *expected* behavior, the *actual* behavior, and, if possible, how to reproduce the behavior.[65] Ideally, the reports should have a consistent form and be stored in a searchable location, such as a bug tracking system. Here, our teams often have customized forms or small web apps that ask for information that's relevant to diagnosing the particular systems they support, which then automatically generate and route a bug. This may also be a good point at which to provide tools for problem reporters to try self-diagnosing or self-repairing common issues on their own.

It's common practice at Google to open a bug for every issue, even those received via email or instant messaging. Doing so creates a log of investigation and remediation activities that can be referenced in the future. Many teams discourage reporting problems directly to a person for several reasons: this practice introduces an additional step of transcribing the report into a bug, produces lower-quality reports that aren't visible to other members of the team, and tends to concentrate the problem-solving load on a handful of team members that the reporters happen to know, rather than the person currently on duty (see also Dealing with Interrupts).

## Shakespeare Has a Problem

You're on-call for the Shakespeare search service and receive an alert, `Shakespeare-BlackboxProbe_SearchFailure`: your black-box monitoring hasn't been able to find search results for "the forms of things unknown" for the past five minutes. The alerting system has filed a bug—with links to the black-box prober's recent results and to the playbook entry for this alert—and assigned it to you. Time to spring into action!

## Triage

Once you receive a problem report, the next step is to figure out what to do about it. Problems can vary in severity: an issue might affect only one user under very specific circumstances (and might have a workaround), or it might entail a complete global outage for a service. Your response should be appropriate for the problem's impact: it's appropriate to declare an all-hands-on-deck emergency for the latter (see Managing Incidents), but doing so for the former is overkill. Assessing an issue's severity requires an exercise of good engineering judgment and, often, a degree of calm under pressure.

Your first response in a major outage may be to start troubleshooting and try to find a root cause as quickly as possible. Ignore that instinct!

Instead, your course of action should be to *make the system work as well as it can under the circumstances*. This may entail emergency options, such as diverting traffic from a broken cluster to others that are still working, dropping traffic wholesale to prevent a cascading failure, or disabling subsystems to lighten the load. Stopping the bleeding should be your first priority; you aren't helping your users if the system dies while you're root-causing. Of course, an emphasis on rapid triage doesn't preclude taking steps to preserve evidence of what's going wrong, such as logs, to help with subsequent root-cause analysis.

Novice pilots are taught that their first responsibility in an emergency is to fly the airplane [Gaw09]; troubleshooting is secondary to getting the plane and everyone on it *safely* onto the ground. This approach is also applicable to computer systems: for example, if a bug is leading to possibly unrecoverable data corruption, freezing the system to prevent further failure may be better than letting this behavior continue.

This realization is often quite unsettling and counterintuitive for new SREs, particularly those whose prior experience was in product development organizations.

## Examine

We need to be able to examine what each component in the system is doing in order to understand whether or not it's behaving correctly.

Ideally, a monitoring system is recording metrics for your system as discussed in Practical Alerting from Time-Series Data. These metrics are a good place to start figuring out what's wrong. Graphing time-series and operations on time-series can be an effective way to understand the behavior of specific pieces of a system and find correlations that might suggest where problems began.[66]

Logging is another invaluable tool. Exporting information about each operation and about system state makes it possible to understand exactly what a process was doing at a given point in time. You may need to analyze system logs across one or many processes. Tracing requests through the whole stack using tools such as Dapper [Sig10] provides a very powerful way to understand how a distributed system is working, though varying use cases imply significantly different tracing designs [Sam14].

### Logging

Text logs are very helpful for reactive debugging in real time, while storing logs in a structured binary format can make it possible to build tools to conduct retrospective analysis with much more information.

It's really useful to have multiple verbosity levels available, along with a way to increase these levels on the fly. This functionality enables you to examine any or all operations in incredible detail without having to restart your process, while still allowing you to dial back the verbosity levels when your service is operating normally. Depending of the volume of traffic your service receives, it might be better to use statistical sampling; for example, you might show one out of every 1,000 operations.

A next step is to include a selection language so that you can say "show me operations that match X," for a wide range of X—e.g., `Set` RPCs with a payload size below 1,024 bytes, or operations that took longer than 10 ms to return, or which called `doSomethingInteresting()` in *rpc_handler.py*. You might even want to design your logging infrastructure so that you can turn it on as needed, quickly and selectively.

Exposing current state is the third trick in our toolbox. For example, Google servers have endpoints that show a sample of RPCs recently sent or received, so it's possible to understand how any one server is communicating with others without referencing an architecture diagram. These endpoints also show histograms of error rates and latency for each type of RPC, so that it's possible to quickly tell what's unhealthy. Some systems have endpoints that show their current configuration or allow examination of their data; for instance, Google's Borgmon servers (Practical Alerting from Time-Series Data) can show the monitoring rules they're using, and even allow tracing a particular computation step-by-step to the source metrics from which a value is derived.

Finally, you may even need to instrument a client to experiment with, in order to discover what a component is returning in response to requests.

## Debugging Shakespeare

Using the link to the black-box monitoring results in the bug, you discover that the prober sends an HTTP GET request to the `/api/search` endpoint:

```
{
      'search_text': 'the forms of things unknown'
    }
```

It expects to receive a response with an HTTP 200 response code and a JSON payload exactly matching:

```
[{
        "work": "A Midsummer Night's Dream",
        "act": 5,
        "scene": 1,
        "line": 2526,
        "speaker": "Theseus"
    }]
```

The system is set up to send a probe once a minute; over the past 10 minutes, about half the probes have succeeded, though with no discernible pattern. Unfortunately, the prober doesn't show you *what* was returned when it failed; you make a note to fix that for the future.

Using `curl`, you manually send requests to the search endpoint and get a failed response with HTTP response code 502 (Bad Gateway) and no payload. It has an HTTP header, `X-Request-Trace`, which lists the addresses of the backend servers responsible for responding to that request. With this information, you can now examine those backends to test whether they're responding appropriately.

## Diagnose

A thorough understanding of the system's design is decidedly helpful for coming up with plausible hypotheses about what's gone wrong, but there are also some generic practices that will help even without domain knowledge.

## Simplify and reduce

Ideally, components in a system have well-defined interfaces and perform known transformations from their input to their output (in our example, given an input search text, a component might return output containing possible matches). It's then possible to look at the connections *between* components—or, equivalently, at the data flowing between them—to determine whether a given component is working properly. Injecting known test data in order to check that the resulting output is expected (a form of black-box testing) at each step can be especially effective, as can injecting data intended to probe possible causes of errors. Having a solid reproducible test case makes debugging much faster, and it may be possible to use the case in a non-production environment where more invasive or riskier techniques are available than would be possible in production.

Dividing and conquering is a very useful general-purpose solution technique. In a multilayer system where work happens throughout a stack of components, it's often best to start systematically from one end of the stack and work toward the other end, examining each component in turn. This strategy is also well-suited for use with data processing pipelines. In exceptionally large systems, proceeding linearly may be too slow; an alternative, *bisection*, splits the system in half and examines the communication paths between components on one side and the other. After determining whether one half seems to be working properly, repeat the process until you're left with a possibly faulty component.

## Ask "what," "where," and "why"

A malfunctioning system is often still trying to do *something*—just not the thing you want it to be doing. Finding out *what* it's doing, then asking *why* it's doing that and *where* its resources are being used or where its output is going can help you understand how things have gone wrong.[67]

### Unpacking the Causes of a Symptom

**Symptom**: A Spanner cluster has high latency and RPCs to its servers are timing out.

**Why**? The Spanner server tasks are using all their CPU time and can't make progress on all the requests the clients send.

**Where** in the server is the CPU time being used? Profiling the server shows it's sorting entries in logs checkpointed to disk.

**Where** in the log-sorting code is it being used? When evaluating a regular expression against paths to log files.

**Solutions**: Rewrite the regular expression to avoid backtracking. Look in the codebase for similar patterns. Consider using RE2, which does not backtrack and guarantees linear runtime growth with input size.[68]

## What touched it last

Systems have inertia: we've found that a working computer system tends to remain in motion until acted upon by an external force, such as a configuration change or a shift in the type of load served. Recent changes to a system can be a productive place to start identifying what's going wrong.[69]

Well-designed systems should have extensive production logging to track new version deployments and configuration changes at all layers of the stack, from the server binaries handling user traffic down to the packages installed on individual nodes in the cluster. Correlating changes in a system's performance and behavior with other events in the system and environment can also be helpful in constructing monitoring dashboards; for example, you might annotate a graph showing the system's error rates with the start and end times of a deployment of a new version, as seen in Figure 12-2.

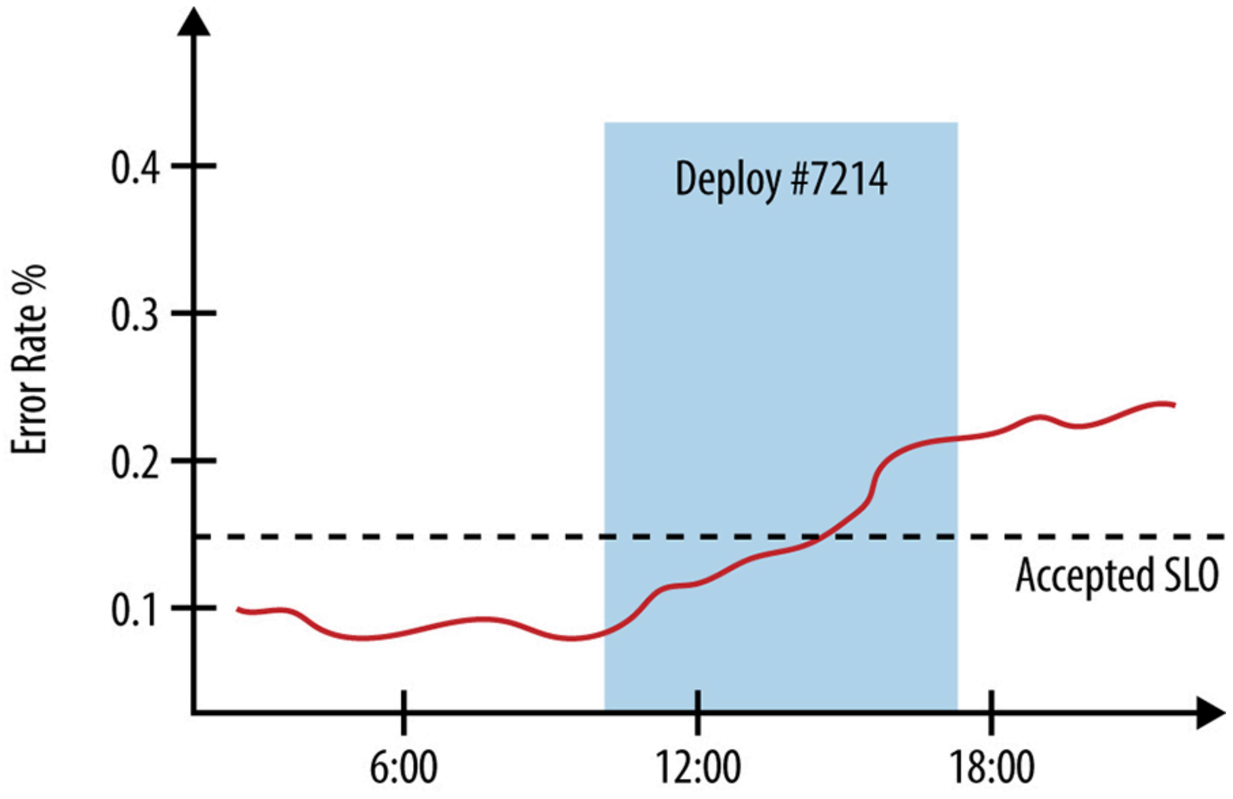

**Figure 12-2. Error rates graphed against deployment start and end times**

Manually sending a request to the `/api/search` endpoint (see Debugging Shakespeare) and seeing the failure listing backend servers that handled the response lets you discount the likelihood that the problem is with the API frontend server and with the load balancers: the response probably wouldn't have included that information if the request hadn’t at least made it to the search backends and failed there. Now you can focus your efforts on the backends—analyzing their logs, sending test queries to see what responses they return, and examining their exported metrics.

## Specific diagnoses

While the generic tools described previously are helpful across a broad range of problem domains, you will likely find it helpful to build tools and systems to help with diagnosing your particular services. Google SREs spend much of their time building such tools. While many of these tools are necessarily specific to a given system, be sure to look for commonalities between services and teams to avoid duplicating effort.

## Test and Treat

Once you’ve come up with a short list of possible causes, it’s time to try to find *which* factor is at the root of the actual problem. Using the experimental method, we can try to rule in or rule out our hypotheses. For instance, suppose we think a problem is caused by either a network failure between an application logic server and a database server, or by the database refusing connections. Trying to connect to the database with the same credentials the application logic server uses can refute the second hypothesis, while pinging the database server may be able to refute the first, depending on network topology, firewall rules, and other factors. Following the code and trying to imitate the code flow, step-by-step, may point to exactly what’s going wrong.

There are a number of considerations to keep in mind when designing tests (which may be as simple as sending a ping or as complicated as removing traffic from a cluster and injecting specially formed requests to find a race condition):

- An ideal test should have mutually exclusive alternatives, so that it can rule one group of hypotheses in and rule another set out. In practice, this may be difficult to achieve.

- Consider the obvious first: perform the tests in decreasing order of likelihood, considering possible risks to the system from the test. It probably makes more sense to test for network connectivity problems between two machines before looking into whether a recent configuration change removed a user’s access to the second machine.

- An experiment may provide misleading results due to confounding factors. For example, a firewall rule might permit access only from a specific IP address, which might make pinging the database from your workstation fail, even if pinging from the application logic server's machine would have succeeded.

- Active tests may have side effects that change future test results. For instance, allowing a process to use more CPUs may make operations faster, but might increase the likelihood of encountering data races. Similarly, turning on verbose logging might make a latency problem even worse and confuse your results: is the problem getting worse on its own, or because of the logging?

- Some tests may not be definitive, only suggestive. It can be very difficult to make race conditions or deadlocks happen in a timely and reproducible manner, so you may have to settle for less certain evidence that these are the causes.

Take clear notes of what ideas you had, which tests you ran, and the results you saw.[70] Particularly when you are dealing with more complicated and drawn-out cases, this documentation may be crucial in helping you remember exactly what happened and prevent having to repeat these steps.[71] If you performed active testing by changing a system—for instance by giving more resources to a process—making changes in a systematic and documented fashion will help you return the system to its pre-test setup, rather than running in an unknown hodge-podge configuration.

# Negative Results Are Magic

Written by Randall Bosetti
Edited by Joan Wendt

A "negative" result is an experimental outcome in which the expected effect is absent—that is, any experiment that doesn't work out as planned. This includes new designs, heuristics, or human processes that fail to improve upon the systems they replace.

**Negative results should not be ignored or discounted.** Realizing you're wrong has much value: a clear negative result can resolve some of the hardest design questions. Often a team has two seemingly reasonable designs but progress in one direction has to address vague and speculative questions about whether the other direction might be better.

**Experiments with negative results are conclusive.** They tell us something certain about production, or the design space, or the performance limits of an existing system. They can help others determine whether their own experiments or designs are worthwhile. For example, a given development team might decide against using a particular web server because it can handle only ~800 connections out of the needed 8,000 connections before failing due to lock contention. When a subsequent development team decides to evaluate web servers, instead of starting from scratch, they can use this already well-documented negative result as a starting point to decide quickly whether (a) they need fewer than 800 connections or (b) the lock contention problems have been resolved.

Even when negative results do not apply directly to someone else's experiment, the supplementary data gathered can help others choose new experiments or avoid pitfalls in previous designs. Microbenchmarks, documented antipatterns, and project postmortems all fit this category. You should consider the scope of the negative result when designing an experiment, because a broad or especially robust negative result will help your peers even more.

**Tools and methods can outlive the experiment and inform future work.** As an example, benchmarking tools and load generators can result just as easily from a disconfirming experiment as a supporting one. Many webmasters have benefited from the difficult, detail-oriented work that produced Apache Bench, a web server loadtest, even though its first results were likely disappointing.

Building tools for repeatable experiments can have indirect benefits as well: although one application you build may not benefit from having its database on SSDs or from creating indices for dense keys, the next one just might. Writing a script that allows you to easily try out these configuration changes ensures you don't forget or miss optimizations in your next project.

**Publishing negative results improves our industry's data-driven culture.** Accounting for negative results and statistical insignificance reduces the bias in our metrics and provides an example to others of how to maturely accept uncertainty. By publishing everything, you encourage others to do the same, and everyone in the industry collectively learns much more quickly. SRE has already learned this lesson with high-quality postmortems, which have had a large positive effect on production stability.

**Publish your results.** If you are interested in an experiment's results, there's a good chance that other people are as well. When you publish the results, those people do not have to design and run a similar experiment themselves. It's tempting and common to avoid reporting negative results because it's easy to perceive that the experiment "failed." Some experiments are doomed, and they tend to be caught by review. Many more experiments are simply unreported because people mistakenly believe that negative results are not progress.

Do your part by telling everyone about the designs, algorithms, and team workflows you've ruled out. Encourage your peers by recognizing that negative results are part of thoughtful risk taking and that every well-designed experiment has merit. Be skeptical of any design document, performance review, or essay that doesn't mention failure. Such a document is potentially either too heavily filtered, or the author was not rigorous in his or her methods.

Above all, publish the results you find surprising so that others—including your future self—aren't surprised.

## Cure

Ideally, you've now narrowed the set of possible causes to one. Next, we'd like to prove that it's the actual cause. Definitively proving that a given factor *caused* a problem—by reproducing it at will—can be difficult to do in production systems; often, we can only find *probable* causal factors, for the following reasons:

- *Systems are complex*. It's quite likely that there are multiple factors, each of which individually is not the cause, but which taken jointly are causes.[72] Real systems are also often path-dependent, so that they must be in a specific state before a failure occurs.

- *Reproducing the problem in a live production system may not be an option*, either because of the complexity of getting the system into a state where the failure can be triggered, or because further downtime may be unacceptable. Having a nonproduction environment can mitigate these challenges, though at the cost of having another copy of the system to run.

Once you've found the factors that caused the problem, it's time to write up notes on what went wrong with the system, how you tracked down the problem, how you fixed the problem, and how to prevent it from happening again. In other words, you need to write a postmortem (although ideally, the system is *alive* at this point!).

## Case Study

App Engine,[73] part of Google's Cloud Platform, is a platform-as-a-service product that allows developers to build services atop Google's infrastructure. One of our internal customers filed a problem report indicating that they'd recently seen a dramatic increase in latency, CPU usage, and number of running processes needed to serve traffic for their app, a content-management system used to build documentation for developers.[74] The customer couldn't find any recent changes to their code that correlated with the increase in resources, and there hadn't been an increase in traffic to their app (see Figure 12-3), so they were wondering if a change in the App Engine service was responsible.

Our investigation discovered that latency had indeed increased by nearly an order of magnitude (as shown in Figure 12-4). Simultaneously, the amount of CPU time (Figure 12-5) and number of serving processes (Figure 12-6) had nearly quadrupled. Clearly something was wrong. It was time to start troubleshooting.

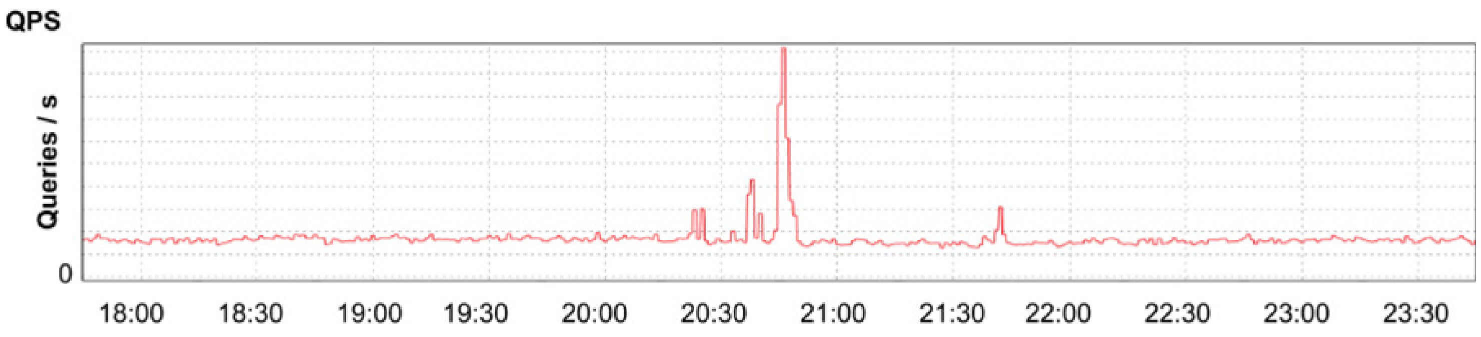

**Figure 12-3. Application's requests received per second, showing a brief spike and return to normal**

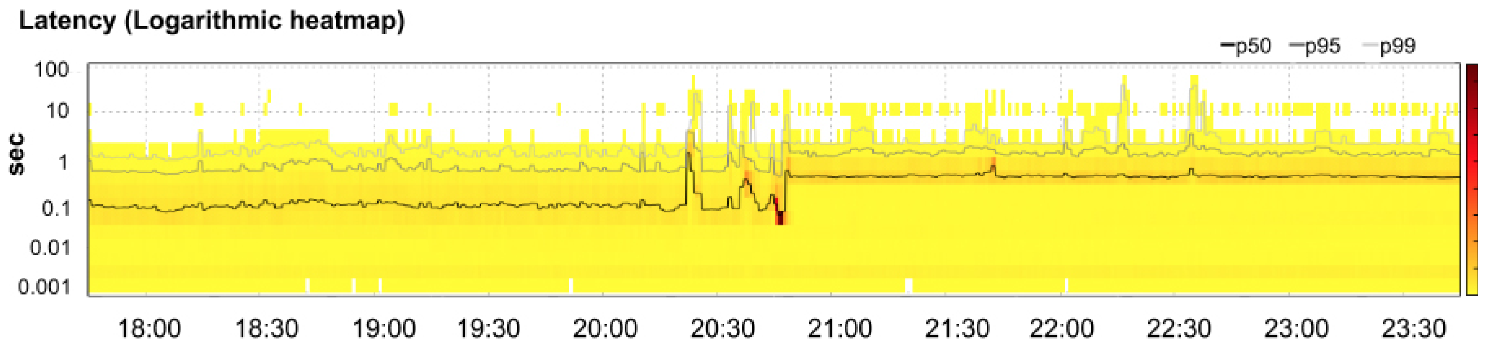

**Figure 12-4. Application's latency, showing 50th, 95th, and 99th percentiles (lines) with a heatmap showing how many requests fell into a given latency bucket at any point in time (shade)**

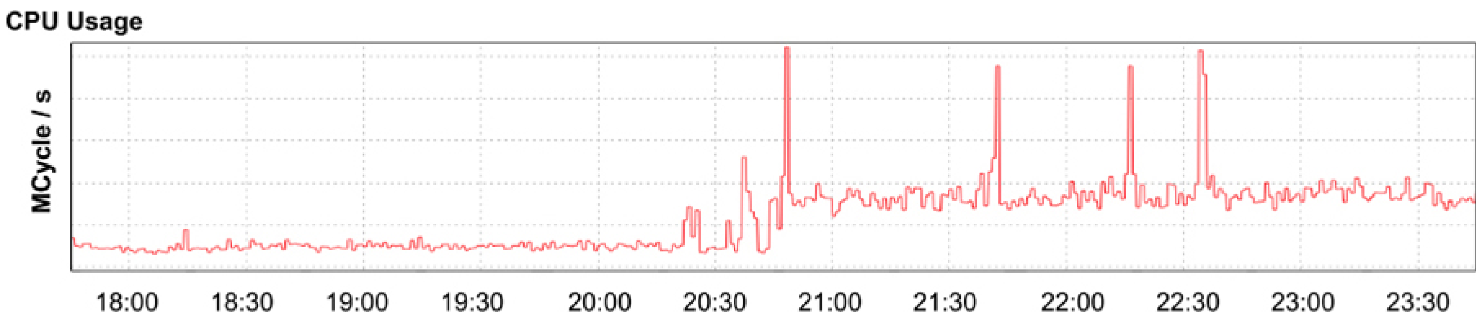

**Figure 12-5. Aggregate CPU usage for the application**

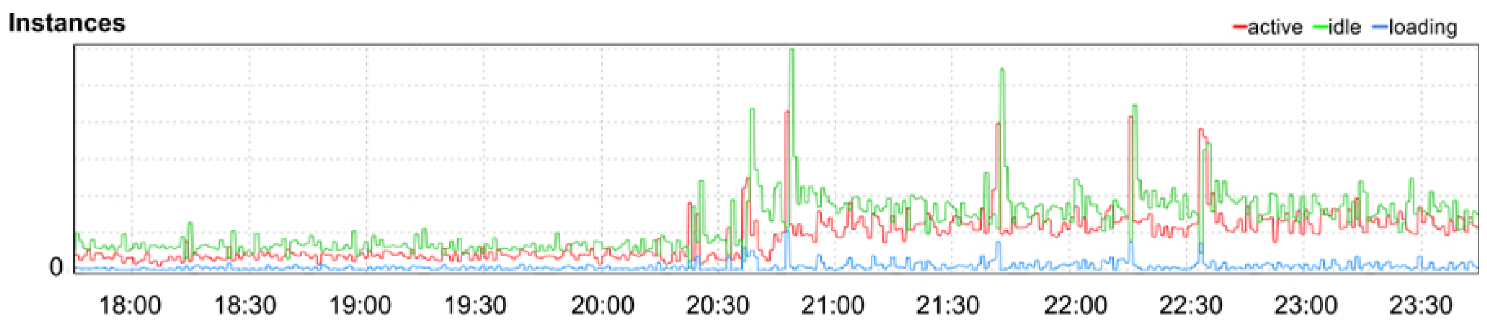

**Figure 12-6. Number of instances for the application**

Typically a sudden increase in latency and resource usage indicates either an increase in traffic sent to the system or a change in system configuration. However, we could easily rule out both of these possible causes: while a spike in traffic to the app around 20:45 could explain a brief surge in resource usage, we'd expect traffic to return to baseline fairly soon after request volume normalized. This spike certainly shouldn't have continued for multiple days, beginning when the app's developers filed the report and we started looking into the problem. Second, the change in performance happened on Saturday, when neither changes to the app nor the production environment were in flight. The service's most recent code pushes and configuration pushes had completed days before. Furthermore, if the problem originated with the service, we'd expect to see similar effects on other apps using the same infrastructure. However, no other apps were experiencing similar effects.

We referred the problem report to our counterparts, App Engine's developers, to investigate whether the customer was encountering any idiosyncrasies in the serving infrastructure. The developers weren't able to find any oddities, either. However, a developer did notice a correlation between the latency increase and the increase of a specific data storage API call, `merge_join`, which often indicates suboptimal indexing when reading from the datastore. Adding a composite index on the properties the app uses to select objects from the datastore would speed those requests, and in principle, speed the application as a whole—but we'd need to figure out *which* properties needed indexing. A quick look at the application's code didn't reveal any obvious suspects.

It was time to pull out the heavy machinery in our toolkit: using Dapper [Sig10], we traced the steps individual HTTP requests took—from their receipt by a frontend reverse proxy through to the point where the app's code returned a response—and looked at the RPCs issued by each server involved in handling that request. Doing so would allow us to see which properties were included in requests to the datastore, then create the appropriate indices.

While investigating, we discovered that requests for static content such as images, which weren't served from the datastore, were also much slower than expected. Looking at graphs with file-level granularity, we saw their responses had been much faster only a few days before. This implied that the observed correlation between `merge_join` and the latency increase was spurious and that our suboptimal-indexing theory was fatally flawed.

Examining the unexpectedly slow requests for static content, most of the RPCs sent from the application were to a memcache service, so the requests should have been very fast—on the order of a few milliseconds. These requests did turn out to be very fast, so the problem didn't seem to originate there. However, between the time the app started working on a request and when it made the first RPCs, there was about a 250 ms period where the app was doing…well, *something*. Because App Engine runs code provided by users, its SRE team does not profile or inspect app code, so we couldn't tell what the app was doing in that interval; similarly, Dapper couldn't help track down what was going on since it can only trace RPC calls, and none were made during that period.

Faced with what was, by this point, quite a mystery, we decided not to solve it…*yet*. The customer had a public launch scheduled for the following week, and we weren't sure how soon we'd be able to identify the problem and fix it. Instead, we recommended that the customer increase the resources allocated to their app to the most CPU-rich instance type available. Doing so reduced the app's latency to acceptable levels, though not as low as we'd prefer. We concluded that the latency mitigation was good enough that the team could conduct their launch successfully, then investigate at leisure.[75]

At this point, we suspected that the app was a victim of yet another common cause of sudden increases in latency and resource usage: a change in the type of work. We'd seen an increase in writes to the datastore from the app, just before its latency increased, but because this increase wasn't very large—nor was it sustained—we'd written it off as coincidental. However, this behavior did resemble a common pattern: an instance of the app is initialized by reading objects from the datastore, then storing them in the instance's memory. By doing so, the instance avoids reading rarely changing configuration from the datastore on each request, and instead checks the in-memory objects. Then, the time it takes to handle requests will often scale with the amount of configuration data.[76] We couldn't prove that this behavior was the root of the problem, but it's a common antipattern.

The app developers added instrumentation to understand where the app was spending its time. They identified a method that was called on every request, that checked whether a user had whitelisted access to a given path. The method used a caching layer that sought to minimize accesses to both the datastore and the memcache service, by holding whitelist objects in instances' memory. As one of the app's developers noted in the investigation, "I don't know where the fire is yet, but I'm blinded by smoke coming from this whitelist cache."

Some time later, the root cause was found: due to a long-standing bug in the app's access control system, whenever one specific path was accessed, a whitelist object would be created and stored in the datastore. In the run-up to launch, an automated security scanner had been testing the app for vulnerabilities, and as a side effect, its scan produced thousands of whitelist objects over the course of half an hour. These superfluous whitelist objects then had to be checked on every request to the app, which led to pathologically slow responses—without causing any RPC calls from the app to other services. Fixing the bug and removing those objects returned the app's performance to expected levels.

## Making Troubleshooting Easier

There are many ways to simplify and speed troubleshooting. Perhaps the most fundamental are:

- Building observability—with both white-box metrics and structured logs—into each component from the ground up
- Designing systems with well-understood and observable interfaces between components.

Ensuring that information is available in a consistent way throughout a system—for instance, using a unique request identifier throughout the span of RPCs generated by various components—reduces the need to figure out *which* log entry on an upstream component matches a log entry on a downstream component, speeding the time to diagnosis and recovery.

Problems in correctly representing the state of reality in a code change or an environment change often lead to a need to troubleshoot. Simplifying, controlling, and logging such changes can reduce the need for troubleshooting, and make it easier when it happens.

## Conclusion

We've looked at some steps you can take to make the troubleshooting process clear and understandable to novices, so that they, too, can become effective at solving problems. Adopting a systematic approach to troubleshooting—as opposed to relying on luck or experience—can help bound your services' time to recovery, leading to a better experience for your users.

---

[58]Indeed, using only first principles and troubleshooting skills is often an effective way to learn how a system works; see Accelerating SREs to On-Call and Beyond.

[59]See *https://en.wikipedia.org/wiki/Hypothetico-deductive_model*.

[60]For instance, exported variables as described in Practical Alerting from Time-Series Data.

[61]Attributed to Theodore Woodward, of the University of Maryland School of Medicine, in the 1940s. See *https://en.wikipedia.org/wiki/Zebra_(medicine)*. This works in some domains, but in some systems, entire classes of failures may be eliminable: for instance, using a well-designed cluster filesystem means that a latency problem is unlikely to be due to a single dead disk.

[62]Occam's Razor; see *https://en.wikipedia.org/wiki/Occam%27s_razor*. But remember that it may still be the case that there are multiple problems; in particular, it may be more likely that a system has a number of common low-grade problems that, taken together, explain all the symptoms rather than a single rare problem that causes them all. Cf *https://en.wikipedia.org/wiki/Hickam%27s_dictum*.

[63]Of course, see *https://xkcd.com/552*.

[64]At least, we have no plausible theory to explain why the number of PhDs awarded in Computer Science in the US should be extremely well correlated ($r^2$ = 0.9416) with the per capita consumption of cheese, between 2000 and 2009: *https://tylervigen.com/view_correlation?id=1099*.

[65]It may be useful to refer prospective bug reporters to [Tat99] to help them provide high-quality problem reports.

[66]But beware false correlations that can lead you down wrong paths!

[67]In many respects, this is similar to the "Five Whys" technique [Ohn88] introduced by Taiichi Ohno to understand the root causes of manufacturing errors.

[68]In contrast to RE2, PCRE can require exponential time to evaluate some regular expressions. RE2 is available at *https://github.com/google/re2*.

[69][All15] observes this is a frequently used heuristic in resolving outages.

[70]Using a shared document or real-time chat for notes provides a timestamp of *when* you did something, which is helpful for postmortems. It also shares that information with others, so they're up to speed with the current state of the world and don't need to interrupt your troubleshooting.

[71]See also Negative Results Are Magic for more on this point.

[72]See [Mea08] on how to think about systems, and also [Coo00] and [Dek14] on the limitations of finding a single root cause instead of examining the system and its environment for causative factors.

[73]See *https://cloud.google.com/appengine*.

[74]We have compressed and simplified this case study to aid understanding.

[75]While launching with an unidentified bug isn't ideal, it's often impractical to eliminate all known bugs. Instead, sometimes we have make do with second-best measures and mitigate risk as best we can, using good engineering judgment.

[76]The datastore lookup can use an index to speed the comparison, but a frequent in-memory implementation is a simple `for` loop comparison across all the cached objects. If there are only a few objects, it won't matter that this takes linear time—but this can cause a significant increase in latency and resource usage as the number of cached objects grows.

PREVIOUS
Chapter 11 - Being On-Call

NEXT
Chapter 13 - Emergency Response

Copyright © 2017 Google, Inc. Published by O'Reilly Media, Inc. Licensed under CC BY-NC-ND 4.0