Lessons Learned from Two Decades of Site Reliability Engineering

Special edition publication commemorating

Google Site Reliability Engineering Team’s 20th Anniversary

Google Site Reliability Engineering

Lessons Learned from Two Decades of Site Reliability Engineering

Or, eleven things we have learned as Site Reliability Engineers at Google

Authors

Adrienne Walcer, Kavita Guliani, Mikel Ward, Sunny Hsiao, and Vrai Stacey

Contributors

Ali Biber, Guy Nadler, Luisa Fearnside, Thomas Holdschick, and Trevor Mattson-Hamilton

Foreword

A lot can happen in twenty years, especially when you're busy growing.

Two decades ago, Google had a pair of small datacenters, each housing a few thousand servers, connected in a ring by a pair of 2.4G network links. We ran our private cloud (though we didn't call it that at the time) using Python scripts such as "Assigner" and "Autoreplacer" and "Babysitter" which operated on config files full of individual server names. We had a small database of the machines (MDB) which helped keep information about individual servers organized and durable. Our small team of engineers used scripts and configs to solve some common problems automatically, and to reduce the manual labor required to manage our little fleet of servers.

Time passed, Google's users came for the search and stayed for the free GB of Gmail, and our fleet and network grew with it. Today, in terms of computing power, we are over 1,000 times as large as we were 20 years ago; in network, over 10,000 times as large, and we spend far less effort per server than we used to while enjoying much better reliability from our service stack. Our tools have evolved from a collection of Python scripts, to integrated ecosystems of services, to a unified platform which offers reliability by default. And our understanding of the problems and failure modes of distributed systems also evolved, as we experienced new classes of outages. We created the Wheel of Misfortune, we wrote Service Best Practices guides, we published Google's Greatest Hits, and today are ~~horrified~~ delighted to present:

Lessons learned from two decades of Site Reliability Engineering

Let’s start back in 2016, when YouTube was offering your favorite videos such as “Carpool Karaoke with Adele” and the ever-catchy “Pen-Pineapple-Apple-Pen.” YouTube experienced a fifteen-minute global outage, due to a bug in YouTube’s distributed memory caching system, disrupting YouTube's ability to serve videos. Here are three lessons we learned from this incident.

1 The riskiness of a mitigation should scale with the severity of the outage

There's a meme where one person posts a picture of a spider seen in their house, and the captain says, "TIME 2 MOVE 2 A NEW HOUSE!". The joke is that the incident (seeing a scary spider) would be responded to with a severe mitigation (abandon your current home and move to a new one). We, here in SRE, have had some interesting experiences in choosing a mitigation with more risks than the outage it’s meant to resolve. During the aforementioned YouTube outage, a risky load-shedding process didn't fix the outage… it instead created a cascading failure.

We learned the hard way that during an incident, we should monitor and evaluate the severity of the situation and choose a mitigation path whose riskiness is appropriate for that severity. In a best case scenario, a risky mitigation resolves an outage. In a worst case scenario, the risky mitigation misfires and the outage is prolonged by something that was intended to fix it. Additionally, if *everything* is broken, you can make an informed decision to bypass standard procedures.

2 Recovery mechanisms should be fully tested before an emergency

An emergency fire evacuation in a tall city building is a terrible opportunity to use a ladder for the first time. Similarly, an outage is a terrible opportunity to try a risky load-shedding process for the first time. To keep your cool during a high-risk and high-stress situation, it's important to practice recovery mechanisms and mitigations beforehand and verify that:

- they'll do what you need them to do
- you know how to do them

Testing recovery mechanisms has a fun side effect of reducing the risk of performing some of these actions. Since this messy outage, we've doubled down on testing.

3 Canary all changes

At one point, we wanted to push a caching configuration change. We were pretty sure that it would not lead to anything bad. But *pretty sure* is not 100% sure. Turns out, caching was a pretty critical feature for YouTube, and the config change had some unintended consequences that fully hobbled the service for 13 minutes. Had we canaried those global changes with a progressive rollout strategy, this outage could have been curbed before it had global impact. Read more about the canary strategy in this paper, and learn more in this video.

Around the same timeframe, YouTube's slightly younger sibling, Google Calendar, also experienced an outage which serves as the backdrop for the next two lessons.

4 Have a "Big Red Button"

A "Big Red Button" is a unique but highly practical safety feature: it should kick off a simple, easy-to-trigger action that reverts whatever triggered the undesirable state to (ideally) shut down whatever's happening. "Big Red Buttons" come in many shapes and sizes—and it's important to identify what those big red buttons might be before you submit a potentially risky action. We once narrowly missed a major outage because the engineer who submitted the would-be-triggering change **unplugged their desktop computer before the change could propagate**. So when planning your major rollouts, consider *What is my big red button?* Ensure every service dependency has a "big red button" to exercise in an emergency. See "Generic Mitigations" for more!

5 Unit tests alone are not enough - integration testing is also needed

Ahh...unit tests. They verify that an individual component can perform the way we need it to. Unit tests have intentionally limited scope, and are super helpful, but they also don't fully replicate the runtime environment and productionized demands that might exist. For this reason, we are big advocates of integration testing! We can use integration tests to verify that jobs and tasks can perform a cold start. Will things work the way we want them to? Will components work together the way we want them too? Will these components successfully create the system we want them to?

This lesson was learned during a Calendar outage in which our testing didn't follow the same path as real use, resulting in plenty of testing... that didn't help us assess how a change would perform in reality.

Shifting to an incident that happened in February 2017, we find our next two lessons.

First, unavailable OAuth tokens caused millions of users to be logged out of devices and services, and 32,000 OnHub and Google WiFi devices to perform a factory reset. Manual account recovery claims jumped by 10x because of failed logins. It took Google about 12 hours to fully recover from the outage.

6 COMMUNICATION CHANNELS! AND BACKUP CHANNELS!! AND BACKUPS FOR THOSE BACKUP CHANNELS!!!

Yes, it was a bad time. You want to know what made it worse? Teams were expecting to be able to use Google Hangouts and Google Meet to manage the incident. But when 350M users were logged out of their devices and services... relying on these Google services was, in retrospect, kind of a bad call. Ensure that you have non-dependent backup communication channels, and that you have tested them.

Then, the same 2017 incident led us to better understand graceful degradation:

7 Intentionally degrade performance modes

It's easy to think of availability as either "fully up" or "fully down", but being able to offer a continuous minimum functionality with a degraded performance mode helps to offer a more consistent user experience. So we've built degraded performance modes carefully and intentionally—so during rough patches, it might not even be user-visible (it might be happening right now!). Services should degrade gracefully and continue to function under exceptional circumstances.

This next lesson is a recommendation to ensure that your last-line-of-defense system works as expected in extreme scenarios, such as natural disasters or cyber attacks, that result in loss of productivity or service availability.

8 Test for Disaster resilience

In December of 2022, there were four long-running submarine cable fiber cuts and two terrestrial fiber cuts in short succession, which reduced the available network capacity to North America by 94% for a few hours. How can we prepare ourselves for such instances?

Besides unit testing and integration testing, there are other types of very important testing: disaster resilience and recovery testing. While resilience testing verifies that your service or system could survive in the event of faults, latency, or disruptions, recovery testing verifies that your service can transition back to homeostasis after a full shutdown. Both should be critical pieces of your business continuity strategy—as described in "Weathering the Unexpected". A useful activity can also be sitting your team down and working through how some of these scenarios could theoretically play out—tabletop game style. This can also be a fun opportunity to explore those terrifying "What Ifs", for example, "What if part of your network connectivity gets shut down unexpectedly?".

9 Automate your mitigations

In March of 2023, a near-simultaneous failure of multiple networking devices occurred in a few datacenters, resulting in a widespread packet loss. In this 6-day outage, an estimated 70% of services experienced varied levels of impact, depending on the location, service load, and configuration at the time of network failure.

In such instances, you can reduce your mean time to resolution (MTTR), by automating mitigating measures done by hand. If there's a clear signal that a particular failure is occurring, then why can't that mitigation be kicked off in an automated way? Sometimes it is better to use an automated mitigation first and save the root-causing for after user impact has been avoided.

10 Reduce the time between rollouts, to decrease the likelihood of the rollout going wrong

In March of 2022, a widespread outage in the payments system prevented customers from completing transactions, resulting in the Pokémon GO community day being postponed. The cause was the removal of a single database field, which should have been safe as all uses of that field were removed from the code beforehand. Unfortunately, a slow rollout cadence of one part of the system meant that the field was still being used by the live system.

Having long delays between rollouts, especially in complex, multiple component systems, makes it extremely difficult to reason out the safety of a particular change. Frequent rollouts—with the proper testing in place— lead to fewer surprises from this class of failure.

11 A single global hardware version is a single point of failure

Having only one particular model of device to perform a critical function can make for simpler operations and maintenance. However, it means that if that model turns out to have a problem, that critical function is no longer being performed.

This happened in March 2020 when a networking device that had an undiscovered zero-day bug, encountered a change in traffic patterns that triggered that bug. As the same model and version of the device was being used across the network, a substantial regional outage ensued. What prevented this from being a total outage was the presence of multiple network backbones that allowed high-priority traffic to be routed via a still working alternative.

Latent bugs in critical infrastructure can lurk undetected until a seemingly innocuous event triggers them. Maintaining a diverse infrastructure, while incurring costs of its own, can mean the difference between a troublesome outage and a total one.

So there you have it! Eleven lessons learned, from two decades of Site Reliability Engineering at Google. Why *eleven*? Well, you see, Google Site Reliability, with our rich history, is still in our *prime*.