Reliability Lessons from Google Cloud

Jonah Mann
April 2020
go/google-cloud-reliability-lessons

This is my recollection of a formative event in my career as an engineer. Please forgive if some of the details are misremembered.

In June 2019, Google Calendar suffered a highly publicized three-hour global outage. In response, Google Cloud leadership decreed that all teams would make reliability their highest priority—above all previously planned work—effective immediately. All non-emergency production releases were frozen while leadership drafted a set of principles defining a sufficient level of safety, and systems would only be allowed to resume releasing once they had met those principles.

After a few days, leadership released their principles. Change surfaces were categorized into one of four "safety levels" (SLs):

0. Surfaces to which changes have immediate global impact
1. Surfaces with canaried rollout of changes
2. Surfaces with *targeted* canarying of changes
3. Surfaces with *automated* targeted canarying of changes

New systems could only be built at SL2 or SL3, and the production freeze would only be lifted for existing systems once they were upgraded to at least SL1.

Committees were formed to flesh out exactly how these safety levels would be defined for different types of systems. (The most common type of system in Google Cloud is an RPC server that serves user-scoped requests.) I had the fortune of serving on the committee for batch processes.

Committees answered questions like: What constitutes canarying for this type of system? What fraction of traffic should be in the canary population? What types of regressions should be monitored for in canary populations? To what extent should such monitoring be automated? How should populations be partitioned for targeted canarying?

For server binary rollouts:

- There should be a canary of 1-10% of traffic that includes a time of peak traffic. IIRC this window had to be at least 20 hours. Note that for servers with a daily release cadence, there were almost always two different builds serving production traffic at the same time. Engineers had to take this into account when making changes.
- Automated canary regression tests monitored the differences between canary and control for every server. For every RPC, the distributions of error rate, latency, and response size should be about the same in the canary cell as in the control cells (unless the changes were intentional). Any service built with the standard Google infrastructure received this monitoring by default.
- Where possible, rollouts should be targeted to begin with the most risk-tolerant traffic and progress to the least risk-tolerant. A typical progression would be (1) a team of QA testers who

perform a set of manual regression tests on critical flows, (2) an opt-in group of teammates and trusted dogfooders, (3) all Google employees, (4) consumer users, (5) enterprise users.
- At all times, there should be an engineer on call capable of performing a rollback, and the rollback mechanism should be tested and shown to work.

Experiment rollouts had the same rollback requirements and should similarly follow a progression from risk-tolerant populations to risk-averse, plus:
- Within each population, experiments should be rolled out exponentially (1%, 5%, 10%, 25%, 50%, 100%) or, for changes impacting database load, linearly (1%, 5%, 10%, 20%, 30%...).

For batch processes:
- Canarying would be accomplished by running the process over progressively more risk-averse partitions, checking that the process behaved as expected before progressing to the next partition. Ideally these checks would be automated, written as a phase of the job. Note that this style of canarying might not be possible for jobs that have to scan an entire corpus, e.g. to look for orphans.
- Mutating jobs should always perform a dry (read-only) run before a live run, and should output an auditable record of all changes made.
- Wherever possible, all storage reads should be from backup replicas rather than live, user-serving storage, and all reads should be filtered at the earliest possible stage.
- Read-only jobs should not have write access to storage.
- It was determined that bad batch processes could be especially destructive, so an extra *restoration* requirement was added. Each batch process should include documentation for how to restore the state before the process ran.
- Many teams made an effort to turn down as many regularly-running mutating batch jobs as possible, converting them to other design patterns such as a persistent queue with one entry per unit of work, where a successful processing of a queue message includes acking the message and reenqueuing it for delivery after a fixed period.

Once the safety levels for each system type were fleshed out, teams all across Google Cloud worked to get their systems upgraded.

This initiative undoubtedly took Google Cloud to a safer, more reliable state than it had previously been in. One aspect, however, that rubbed me the wrong way was the top-down nature of the initial reaction to the outages. Engineers were told to stop working on projects that they were emotionally invested in, and projects approaching launch had bureaucratic procedural requirements abruptly changed at the last minute. The cost of these mandates on employee morale was not taken into account.

Furthermore, from my perspective, leadership had long been neglecting rank-and-file engineers' complaints that reliability took a back seat in performance reviews and promotion decisions to feature launches, a problem which I don't think any sincere initiative to improve reliability could be complete without fixing.