A Collection of Best Practices for Production Services

Written by Ben Treynor Sloss
Edited by Betsy Beyer

Fail Sanely

Sanitize and validate configuration inputs, and respond to implausible inputs by *both* continuing to operate in the previous state *and* alerting to the receipt of bad input. Bad input often falls into one of these categories:

Incorrect data

> Validate both syntax and, if possible, semantics. Watch for empty data and partial or truncated data (e.g., alert if the configuration is N% smaller than the previous version).

Delayed data

> This may invalidate current data due to timeouts. Alert well before the data is expected to expire.

Fail in a way that preserves function, possibly at the expense of being overly permissive or overly simplistic. We've found that it's generally safer for systems to continue functioning

with their previous configuration and await a human's approval before using the new, perhaps invalid, data.

Examples

In 2005, Google's global DNS load- and latency-balancing system received an empty DNS entry file as a result of file permissions. It accepted this empty file and served `NXDOMAIN` for six minutes for all Google properties. In response, the system now performs a number of sanity checks on new configurations, including confirming the presence of virtual IPs for *google.com*, and will continue serving the previous DNS entries until it receives a new file that passes its input checks.

In 2009, incorrect (but valid) data led to Google marking the entire Web as containing malware [May09]. A configuration file containing the list of suspect URLs was replaced by a single forward slash character (/), which matched all URLs. Checks for dramatic changes in file size and checks to see whether the configuration is matching sites that are believed unlikely to contain malware would have prevented this from reaching production.

Progressive Rollouts

Nonemergency rollouts *must* proceed in stages. Both configuration and binary changes introduce risk, and you mitigate this risk by applying the change to small fractions of traffic and capacity at one time. The size of your service or rollout, as well as your risk profile, will inform the percentages of production capacity to which the rollout is pushed, and the appropriate time frame between stages. It's also a good idea to perform different stages in

different geographies, in order to detect problems related to diurnal traffic cycles and geographical traffic mix differences.

Rollouts should be supervised. To ensure that nothing unexpected is occurring during the rollout, it must be monitored either by the engineer performing the rollout stage or—preferably—a demonstrably reliable monitoring system. If unexpected behavior is detected, roll back first and diagnose afterward in order to minimize Mean Time to Recovery.

Define SLOs Like a User

Measure availability and performance in terms that matter to an end user. See Service Level Objectives for more discussion.

Example

Measuring error rates and latency at the Gmail client, rather than at the server, resulted in a substantial reduction in our assessment of Gmail availability, and prompted changes to both Gmail client and server code. The result was that Gmail went from about 99.0% available to over 99.9% available in a few years.

Error Budgets

Balance reliability and the pace of innovation with error budgets (see Motivation for Error Budgets), which define the acceptable level of failure for a service, over some period; we often use a month. A budget is simply 1 minus a service's SLO; for instance, a service with a 99.99% availability target has a 0.01% "budget" for unavailability. As long as the service hasn't spent its error budget for the month through the background rate of errors plus any downtime, the development team is free (within reason) to launch new features, updates, and so on.

If the error budget is spent, the service freezes changes (except urgent security and bug fixes addressing any cause of the increased errors) until either the service has earned back room in the budget, or the month resets. For mature services with an SLO greater than 99.99%, a quarterly rather than monthly budget reset is appropriate, because the amount of allowable downtime is small.

Error budgets eliminate the structural tension that might otherwise develop between SRE and product development teams by giving them a common, data-driven mechanism for assessing launch risk. They also give both SRE and product development teams a common goal of developing practices and technology that allow faster innovation and more launches without "blowing the budget."

Monitoring

Monitoring may have only three output types:

Pages

> A human must do something *now*

Tickets

> A human must do something within a few days

Logging

> No one need look at this output immediately, but it's available for later analysis if needed

If it's important enough to bother a human, it should either *require* immediate action (i.e., page) or be treated as a bug and entered into your bug-tracking system. Putting alerts into email and hoping that someone will read all of them and notice the important ones is the moral equivalent of piping them to */dev/null*: they will eventually be ignored. History demonstrates this strategy is an attractive nuisance because it can work for a while, but it

relies on eternal human vigilance, and the inevitable outage is thus more severe when it happens.

Postmortems

Postmortems (see Postmortem Culture: Learning from Failure) should be blameless and focus on process and technology, not people. Assume the people involved in an incident are intelligent, are well intentioned, and were making the best choices they could given the information they had available at the time. It follows that we can't "fix" the people, but must instead fix their environment: e.g., improving system design to avoid entire classes of problems, making the appropriate information easily available, and automatically validating operational decisions to make it difficult to put systems in dangerous states.

Capacity Planning

Provision to handle a simultaneous planned and unplanned outage, without making the user experience unacceptable; this results in an "N + 2" configuration, where peak traffic can be handled by N instances (possibly in degraded mode) while the largest 2 instances are unavailable:

- Validate prior demand forecasts against reality until they consistently match. Divergence implies unstable forecasting, inefficient provisioning, and risk of a capacity shortfall.

- Use load testing rather than tradition to establish the resource-to-capacity ratio: a cluster of X machines could handle Y queries per second three months ago, but can it still do so given changes to the system?

- Don't mistake day-one load for steady-state load. Launches often attract more traffic, while they're also the time you especially want to put the product's best

foot forward. See Reliable Product Launches at Scale and Launch Coordination Checklist.

Overloads and Failure

Services should produce reasonable but suboptimal results if overloaded. For example, Google Search will search a smaller fraction of the index, and stop serving features like Instant to continue to provide good quality web search results when overloaded. Search SRE tests web search clusters beyond their rated capacity to ensure they perform acceptably when overloaded with traffic.

For times when load is high enough that even degraded responses are too expensive for all queries, practice graceful load shedding, using well-behaved queuing and dynamic timeouts; see Handling Overload. Other techniques include answering requests after a significant delay ("tarpitting") and choosing a consistent subset of clients to receive errors, preserving a good user experience for the remainder.

Retries can amplify low error rates into higher levels of traffic, leading to cascading failures (see Addressing Cascading Failures). Respond to cascading failures by dropping a fraction of traffic (including retries!) upstream of the system once total load exceeds total capacity.

Every client that makes an RPC must implement exponential backoff (with jitter) for retries, to dampen error amplification. Mobile clients are especially troublesome because there may be millions of them and updating their code to fix behavior takes a significant amount of time—possibly weeks—and requires that users install updates.

SRE Teams

SRE teams should spend no more than 50% of their time on operational work (see Eliminating Toil); operational overflow should be directed to the product development team. Many services also include the product developers in the on-call rotation and ticket handling, even if there is currently no overflow. This provides incentives to design systems that

minimize or eliminate operational toil, along with ensuring that the product developers are in touch with the operational side of the service. A regular production meeting between SREs and the development team (see Communication and Collaboration in SRE) is also helpful.

We've found that at least eight people need to be part of the on-call team, in order to avoid fatigue and allow sustainable staffing and low turnover. Preferably, those on-call should be in two well-separated geographic locations (e.g., California and Ireland) to provide a better quality of life by avoiding nighttime pages; in this case, six people at each site is the minimum team size.

Expect to handle no more than two events per on-call shift (e.g., per 12 hours): it takes time to respond to and fix outages, start the postmortem, and file the resulting bugs. More frequent events may degrade the quality of response, and suggest that something is wrong with (at least one of) the system's design, monitoring sensitivity, and response to postmortem bugs.

Ironically, if you implement these best practices, the SRE team may eventually end up out of practice in responding to incidents due to their infrequency, making a long outage out of a short one. Practice handling hypothetical outages (see Disaster Role Playing) routinely and improve your incident-handling documentation in the process.

← PREVIOUS
Appendix A - Availability Table

NEXT
Appendix C - Example Incident State Document