generic mitigations for great justice

written by macey | reviewed by stepand, jlbec
created 2018-12-05 | go/generic-mitigations | externally available

Your service should have at least one or two generic mitigations. If it doesn't, you're in for a bad time. If it does - treasure them, maintain them, and **use** them, lest they rot beneath your feet.

hold up, hang on, what's a generic mitigation?

Alrighty. From the top: a mitigation is any action you might take to reduce the impact of a breakage, generally in production. A hotfix is a mitigation. SSHing into an instance and clearing the cache - mitigation. Turning off the machines to close down a vulnerability - mitigation, I guess, in the same way a guillotine cures the common cold.

A **generic mitigation**, then, is one which is useful in mitigating a wide variety of outages.

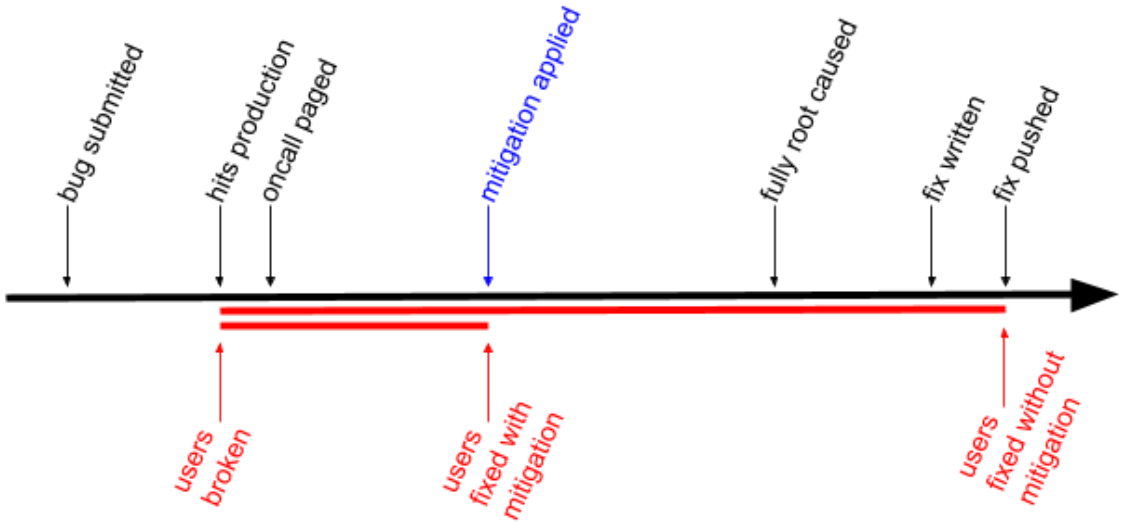
A binary rollback is the most common example of a generic mitigation. Many multi-homed services will have some form of 'drain the service from this location' - a good traffic-based generic mitigation. Others might have separate data rollbacks, or a button to quickly add a whole bunch more capacity.

The most important characteristic of a generic mitigation is this: **You don't need to fully understand your outage to use it.**

I mean, don't we want to understand our outages?

No.

Okay, let me expand on that: you want to understand your outage **after it is mitigated**. Let's draw a quick picture.



The goal of building good generic mitigations is to have a weapon to wield as early in the timeline as possible. If the only mitigations available to you are problem-specific, then you will be unable to help your users until you understand the problem in detail. It is very hard to decrease the time it takes to understand a problem - if we could easily find problems, then we most likely wouldn't have caused them.

So let's assume that the measure of success for a team's incident management is not 'time to fix', but 'time to mitigate' - let's assume the thing we want to minimise is the time our users spend broken. **It is far easier to build mitigations with broad application than it is to make root-causing faster.**

sure. I'm with you. what's stopping us?

Ironically? **Generic mitigations are highly specific.**

That's a nonsense sentence. Let's try again. A good generic 'something terrible is happening, quick, take action' button is simple (and safe!) to use in an emergency; this takes a lot of work beforehand to achieve, and needs to be carefully tailored to the service it is mitigating.

There are, however, some very good **general patterns** for generic mitigations which we can learn from, though these will need careful tweaking to make them fit your specific service.

okay, I'll bite. what're some generic mitigation patterns?

Glad you asked! Here's a handful to get you started.

Drain	Move your traffic to a different place. This one works great if you're multi-homed and traffic driven: if one cell sees high errors, move the queries elsewhere. Most common pattern here is gslb drain, which is used across google3 borg jobs.
Hospitalize	A good way to address the 'bad instance' problem; named for Bigtable's hospital server. This pattern involves having a way to isolate a given unit of your service - a row, a user, a traffic stream - so whatever bug it is hitting stops breaking others.
Rollback	Does what it says on the tin. Reverts your service - most commonly, your binary - to a known-good state. Virtually every service can implement this strategy. Far too many *think* they have rollbacks, only to discover, during an outage, that they don't.
Data rollback	Sub-case of the previous one - just reverts your data. Content-heavy services are more likely to find this useful, particularly those building data from pipelines.
Degrade	Is your service overloaded? Having a way to do less work but stay up is a huge improvement over crashing. Attempts to do this on the fly will always bite you.
Upsize	Got too much traffic? Or is everything running hot for no reason? Add more replicas. It's expensive, but cheaper than pissing off users with an outage. Note that this is rarely as simple as 'scale up one binary'; system scaling is complex.
Block list	Got a query of death? Single spammy user taking out a zone? Just block them.

those... aren't really all that generic?

Yeah, you'd look pretty silly if you drained a cluster because a recent release there triggered errors, only to see the same errors 30 minutes later in the next cluster to upgrade. None of these are magic: you need to diagnose to the level of understanding the *kind* of failure scenario.

They're also not all useful to all services. Draining is nonsense for a lot of cloud products, whose users are pinned to a single physical zone. Hospitalizing is no use when you're multi single tenant or just very well isolated to start with. Degraded mode isn't always an option.

So ask yourself what strategies make sense for your system. Make that part of your production readiness review[1] - write down existing generic mitigations, and plan to add more if you're falling short. The better and more standardised your mitigations, the easier it is to scale support - there'll be no need to page an expert awake at 2am if the mitigation will hold 'til the morning.

Also? If you don't use them - don't drill them, test them, make a little checklist so new team members practice running them against your staging environment before going oncall - they won't work. Tape backups you've never restored from are just really expensive paperweights. Our systems are complex enough that you **cannot know** the knock-on effects of actually using your generic mitigation until you've used it in anger.

The goal is to create easy, frictionless panic buttons for your service that any oncall feels comfortable using at the slightest suspicion they might help. That's not easy to build. But it is far, far cheaper to build them in advance than the lengthy, unmitigatable outages you'll see without.

what's your point? this is a lot of words, yo.

Okay, fine fine. In brief: **The most expensive stretch of an outage is the time when users can see it.** Forget fixing broken things; forget building the perfect system which will never break. A quick path to 'mostly working' is far better for the user than waiting for 'perfectly fixed.' If you're driving a majestic old Toyota Corolla, sometimes the best tactic is making sure there's duct tape in the glove box so you can make it to the mechanic when the wing mirror falls off.

Know what your duct tape is. You're gonna need it.

[1] go/prr

thoughts?

gUsername/Date	Comment
<your details>	<your comment>
lutzky/2019-11-25	This document ages like fine beverage; as the Search stack gets more complicated and the code yellows become, uh, yellower, the more we find ourselves needing this. In go/searchoncall#rolling-back-several-datapushes-at-once, we've added a button for "roll back all 26 datapushes that are loosely connected to some aspect of the search stack".
pstoneman/2019-01-07	+1 to comments above – Nice doc. I'd also note that SRE will be likely to apply one or more of the generic mitigations when they're paged, even without fully understanding the problem or reading the playbook in detail, or whatever. That means that we'd very much like dev teams to make whichever generic mitigations apply to our service to be a) not harmful (i.e. graceful failover/lameducking), and b) effective (i.e. pay attention to the drain signal for serving, for master election, etc etc).
dagitses/2019-01-08	Loved reading this. It's very valuable to have put a name on this practice so common, even a SWE can identify it. Now that we have a name for this concept, we can talk about it more easily and ensure that our services and products have generic mitigations. Thanks!
youngman/2019-01-11	Thanks. This was a very worthwhile read. One point it might be worth emphasising is that teams should not only familiarise themselves with how to do each of these mitigations, but should have a good understanding of the time they take. One of my services for example, a (global) binary rollback would take more than an hour, while a drain takes only minutes. Choosing the wrong mitigation strategy can hugely extend the duration of the visible part of the outage.
mgflax/2018-02-18	Thanks. Another generic mitigation is the "rolling restart" (i.e. "have you tried turning it off and on again...")