Dude, you forgot the feedback

How your open loop control planes are causing outages

Laura de Vesine
silverrose@datadoghq.com

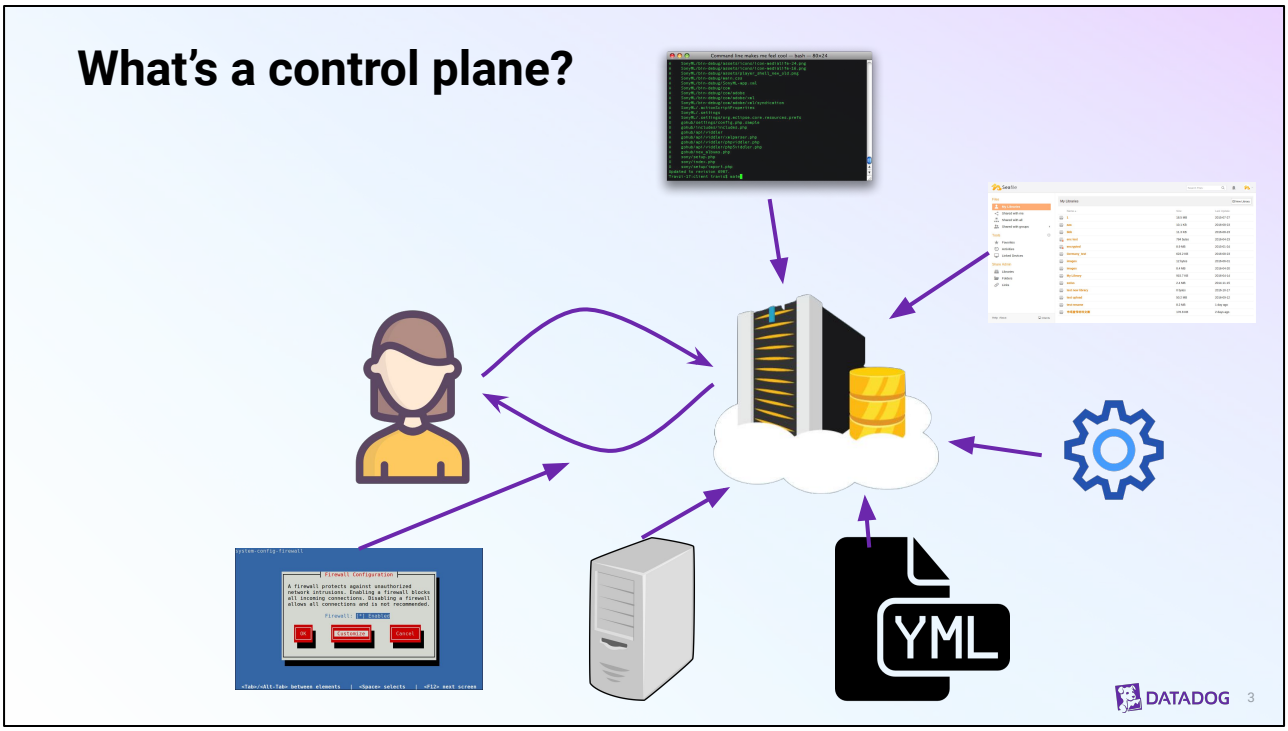
DATADOG

What's is it we *do* here?

DATADOG 2

Good morning SREcon! Since I'm lucky enough to start us off, I'm going to be the first to ask a question you'll probably find folks thinking about a lot this week: What would you say ya *do* here? That is, what does it mean to be an SRE? This is a question I've seen us asking ourselves quite a bit over the last few years. There's a lot of answers here, but *one* answer, and the one I'd like to focus on for this talk, is that SREs are the people who specialize in building and using control planes in complex systems. And y'all, I have some bad news: we *suck* at it. Building good control planes is hard, and using bad ones is also very hard. I'm going to use this time in front of you to share one of the ways I see us building control planes that could be so much better: we forget to design in feedback, and set ourselves and others up for failure.

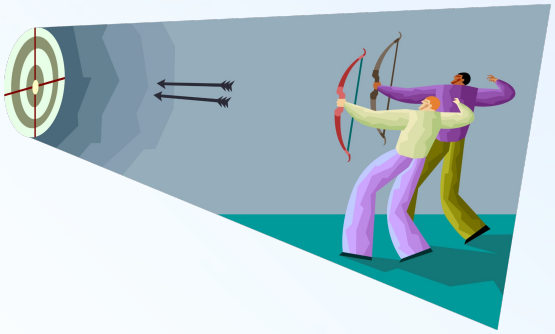
Image: "The Bobs," Office Space (1999)



Since it's kind of central to my point here, let's make sure we're on the same page about what a "control plane" is (whether the idea of a control plane is already familiar to you or not). As a general rule, what software does is take requests from users, and give answers back, right? Software that we think of as having that primary job is a "data plane".
And then we have <click> all the many, many things we use to configure and operate that data plane software – everything from configuration of the software itself, to setting up (and configuring!) the machines it runs on and its network. All of that management stuff is our "control plane". It's the things we use to make sure our software is operating well, also known as "all the stuff SREs build and interact with". This is our bread and butter; the main thing we do and interact with in the typical "SRE" job.

Server in cloud Image by https://www.vectorportal.com
User Image by https://icon-icons.com/icon/girl-person-woman-people/51109
Seafile UX image from
https://en.wikipedia.org/wiki/File:Seafile_6.0_web_interface.png
Gear icon from
https://icon-icons.com/icon/tool-tools-setting-configuration-preferences-options-settings-gear-cog/221263
Server image https://reserve.freesvg.org/server-icon-vector-image
YML file image
https://icon-icons.com/icon/yml-document-black-interface-symbol/57616
Firewall configuration image https://www.flickr.com/photos/xmodulo/15628237745

UX feedback is important



DATADOG 4

And given that we build and interact a lot with this control plane stuff, let’s look at the user experience we’re building for ourselves
A metaphor UX designers (and others) sometimes use is that it's easier to aim a garden hose than an arrow. The reason is that a garden hose comes with continuous feedback about your aim, allowing you to make small adjustments with good information instead of guessing or overshooting. Shooting targets with an arrow is a literal olympic sport; shooting targets with a hose is something virtually anyone can do. Yet we routinely design the control planes in our systems to be an "arrow style" UX: send a command, maybe get an ack that the command was received, then no feedback about what's actually happening in production. We can (and do) learn to work around this, but it doesn’t set us up for success. That’s especially relevant when not everyone interacting with our control planes is a paranoid, grizzled SRE.

Archery image: https://www.wannapik.com/vectors/16697
Hose emoji AI generated

Patterns in postmortems

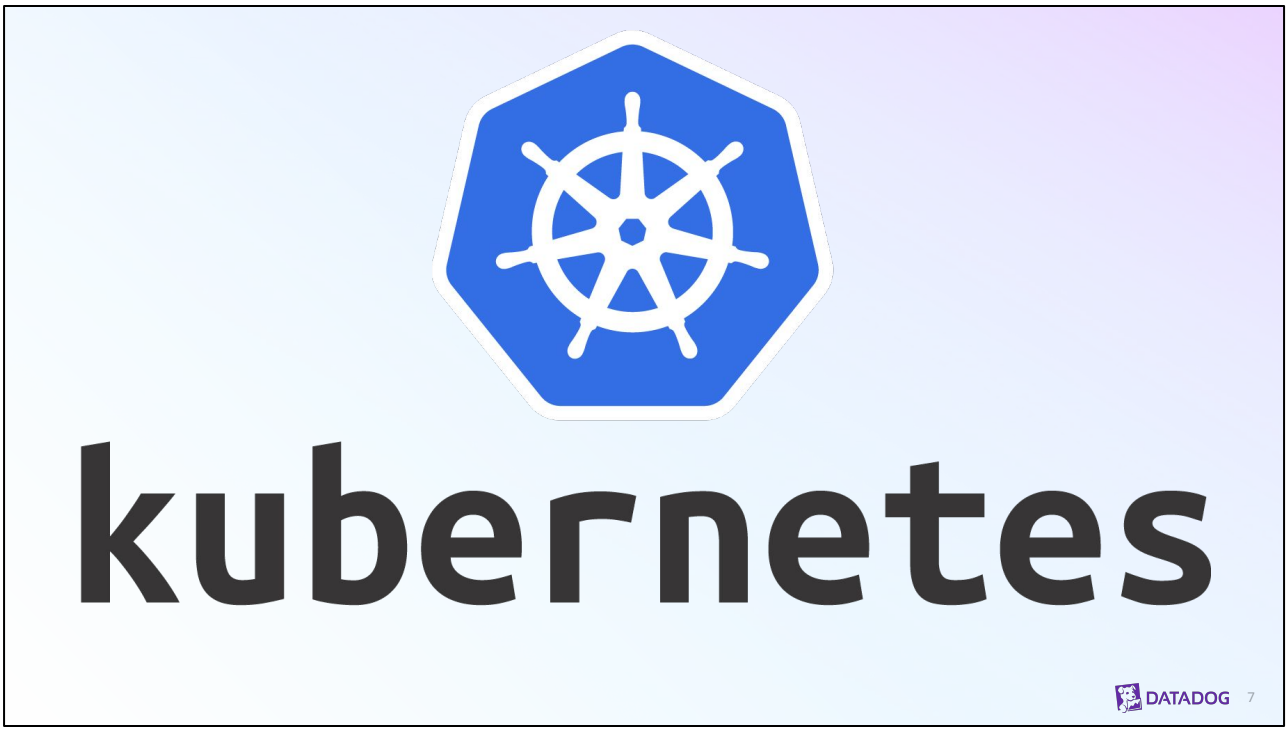
- **Accidentally** <action>ed
- <Action> **unexpectedly** resulted in <system change>
- **Did not realize** that…
- **Intended** to take <action>, but **instead**…

Looking for

- Surprise
- Misfired intent
- Lack of situational awareness

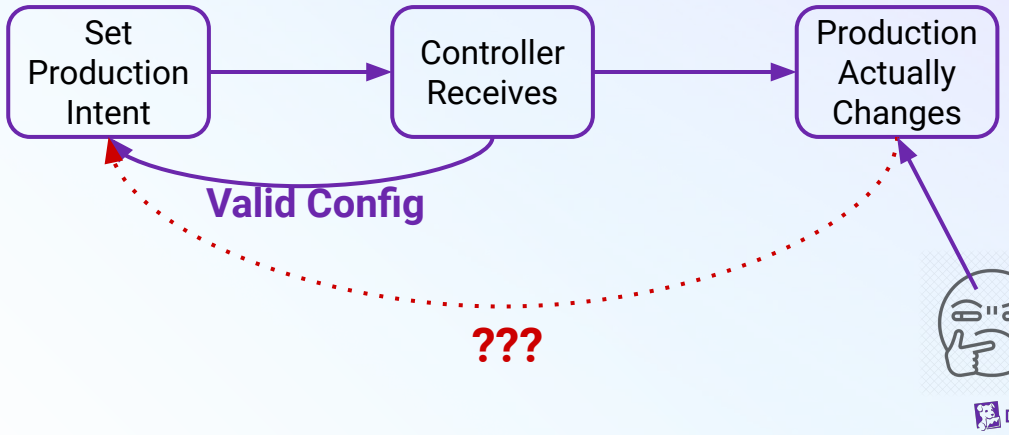
<slide appears by para>
Let me be clear – When I say we're not setting ourselves up for success, I mean we're causing outages. In the spirit of true blamelessness – if the person working with the control plane took the "wrong" action and caused (or worsened) an outage, the problem is the control plane and its UX, *not the operator*. If you're looking for it, you can spot this when it's happened in a postmortem with phrases like (slide). What you're looking for is comments that indicate an operator *did not expect* that the action they took would have the effect on the system that it did. "How did our control plane lead you astray" is a very useful question to ask as part of a blameless postmortem to help find these cases!

So… this matters because stuff breaks, and everyone loves a good outage story. Let’s talk about some!



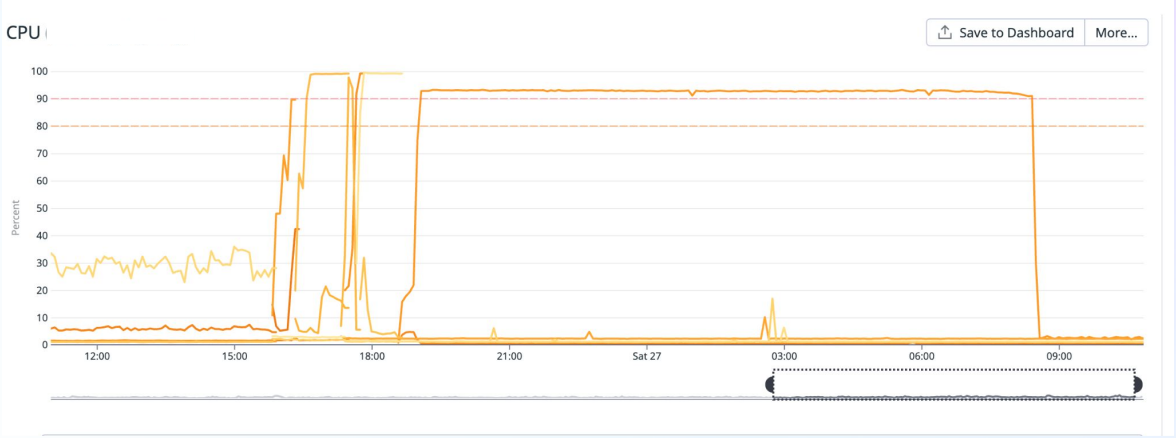
First off, let's talk about my very favorite example of a terrible control plane with bad feedback (wait for hopefully laughing)

Okay well really “intent based” systems



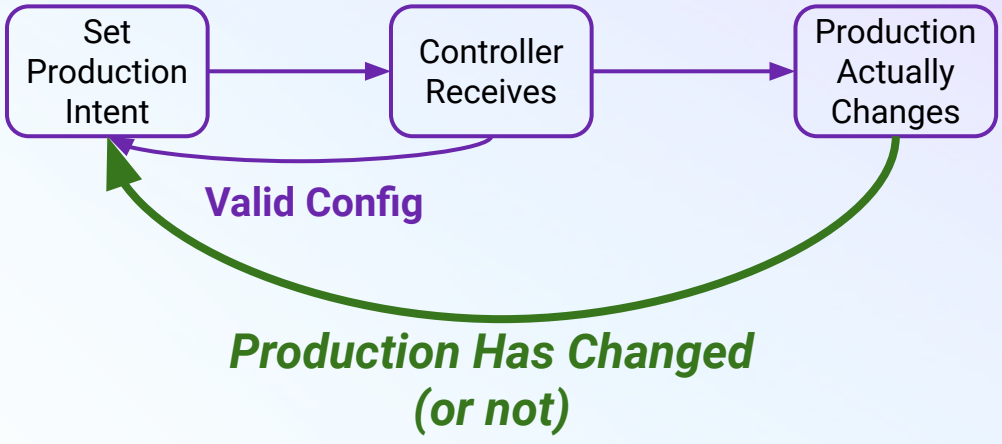
Okay, really I have this complaint about all intent-based control plane systems. These are systems where you tell the control plane what you want (say, “I want 200 pods with this binary and size running”), <click> it accepts your desire and you get feedback <click> about whether that’s valid and well-formed configuration. Then <click> the controller changes production to match your intent…….. <click> well, probably. How do you know production actually changed? Where is that feedback loop? What if there’s a problem (like there aren’t enough nodes to run the pods on, or they’re failing to start up, or they never become healthy, or…). <click> Certainly you *can* find out about those problems, but you have to go ask in another place deliberately, or build that monitoring on top of the as-designed control plane and add your own feedback.

How it goes wrong



I've already given an example of how this goes wrong – you try to scale up to prevent or mitigate an outage, and something goes wrong with that scale-up. Watching this happen in real outages, what I see responders do is to watch the graph looking for CPU load to come down. At some point, they've been waiting for long enough without a fix that they realize something must be wrong. At that point they go check the graph for number of healthy pods running, or query the k8s API for the state of pods starting up (or both). But because they have to actively seek out the information about the actual state of production and anything blocking it from reaching the intended state, there's a delay in the response to the incident. I will definitely bet money that most of the folks in this room have seen a version of this incident.

How to fix it



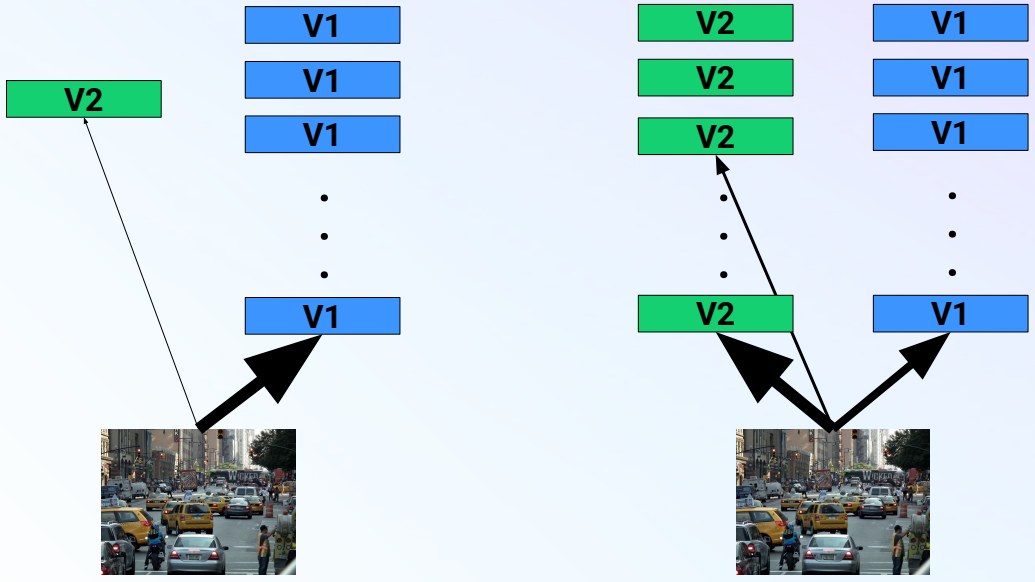
And the fix for this is obvious – build our tooling so that even if it's intent-based, after a change to production is requested, there's automated polling and direct feedback about whether that state has been realized, and notifications when there's a problem. The most trivially simple version of this is something like having the CLI give a "watch your pods start up at this link" message in response to a command, but more active feedback (while being careful of noise) is usually better here.

Fire-at-production

In a similar vein, we have what I think of as the “fire-at-production” control pattern. It’s similar to “intent based” with the relatively subtle difference being that you’re specifying a particular action, not an end state… but you’re still not getting any feedback on the results of your action. This pattern is really common in release systems in my experience – you (or automation) specify “deploy x build”, the release system kicks off, and then…. is it on the machines yet? Are those machines behaving differently based on your change? (not just “have they broken”, but “did the change have an effect”).

Dart image from https://openclipart.org/detail/228817/blindfolded-darts-player

Outage: unexpected out-of-sequence deployments



DATADOG 12

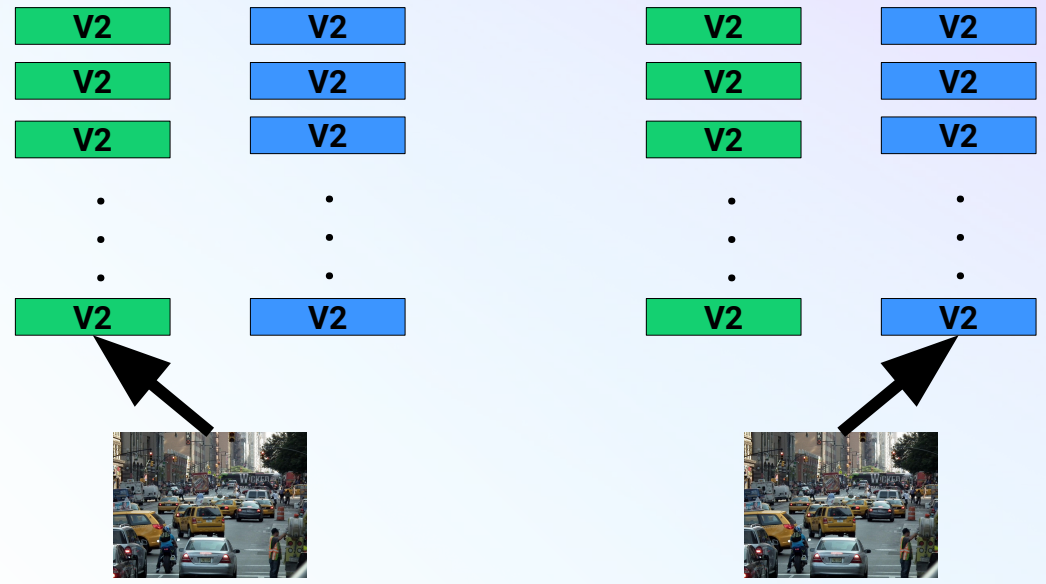
And it matters what the direct effect is of the deployment and where it's at!
We have a service that has implemented blue/green deployments. The particular order for this service's deployment is:

(1) scale down the green/"canary" instance to a small number of pods, roll out the new version to it, and send it a small % of traffic <click>

(2) gradually scale the green instance up, sending progressively more traffic, until it is receiving 100% of traffic <click "animation">

traffic photo from https://www.flickr.com/photos/edrost88/6279776820

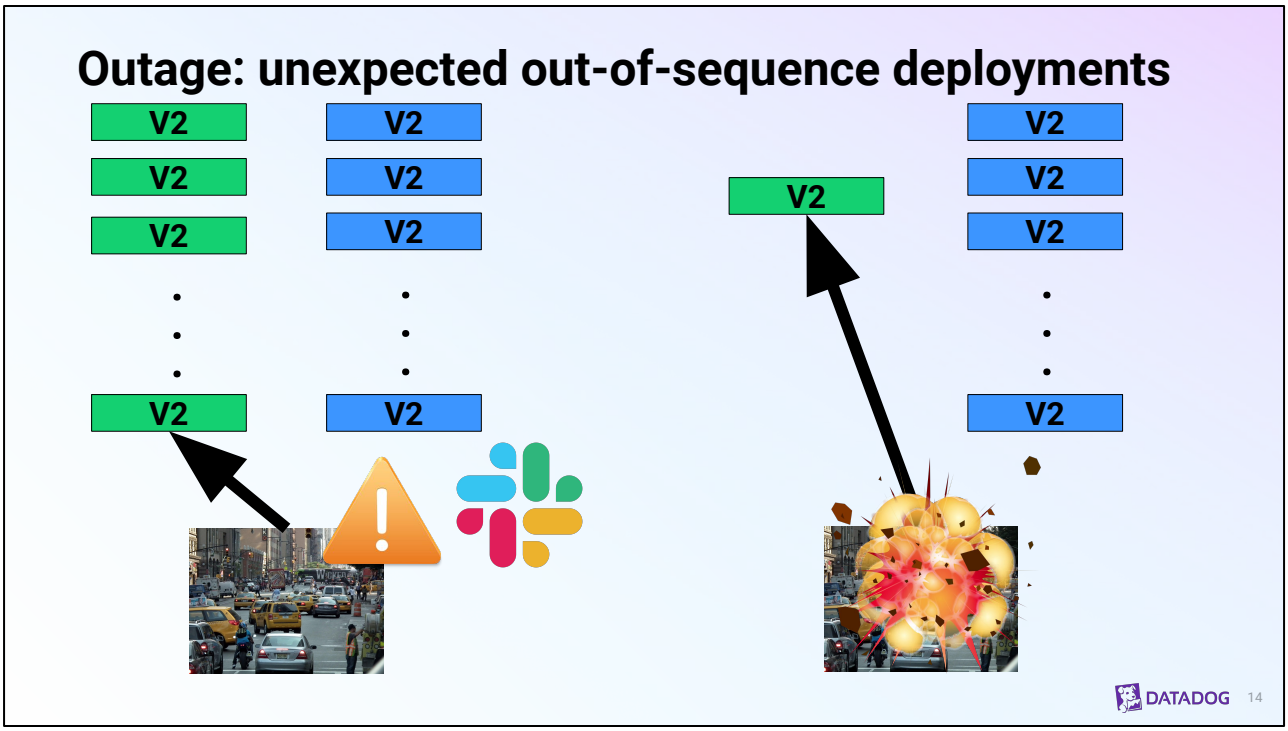
Outage: unexpected out-of-sequence deployments



(3) roll out the new version to the “blue” instance at full scale <click>
(4) send 100% of traffic to the blue instance

You might already see where this is going. <click>

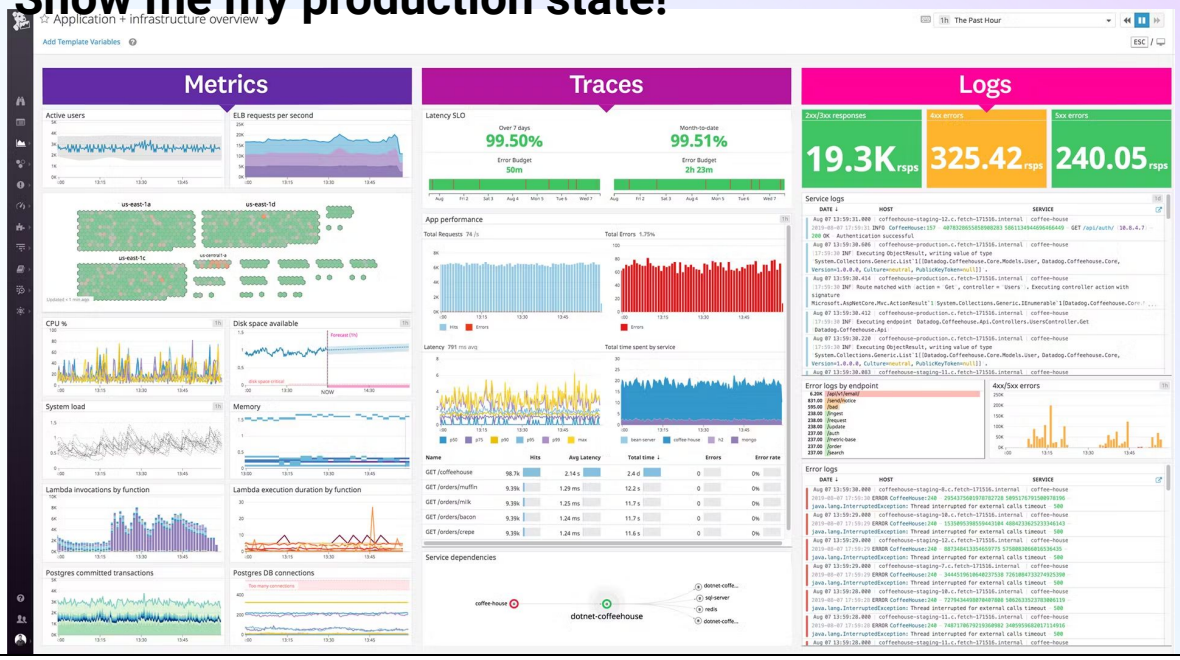
traffic photo from https://www.flickr.com/photos/edrost88/6279776820



Some automation paused the rollout before the cut was made back to the blue instance, and tried (in Slack) to tell a human there was an issue that needed approval before completing. No human noticed the prompt, so the rollout timed out. The system was working fine – all traffic was going to the green instance, which had the new version, and was scaled up to handle the load.
And then a new rollout was started <click>… and the first thing the new rollout does is scale the green instance to a single pod. <click> Oops.

traffic photo from https://www.flickr.com/photos/edrost88/6279776820
alert icon from https://icon-icons.com/icon/alert/14536
explosion from openclipart

Show me my production state!



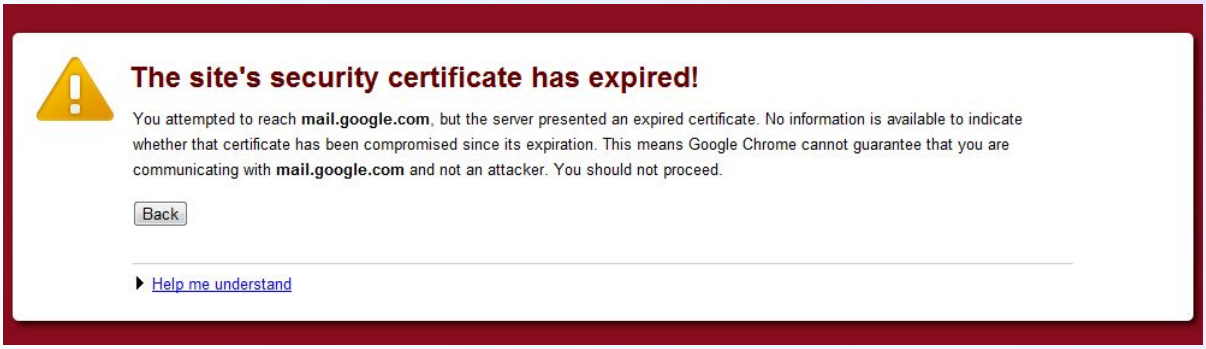
You need to know the *actual state of production* in the *same place* where you're about to take an action on it. The rollout UI needs to show not just the progression of the rollout, but how many instances are running each version, and how traffic is being load balanced. It needs to be built into the scale down action to make sure a human actually sees that traffic is going to the pods about to be scaled down, and how much. And the next rollout shouldn't be able to start without someone being aware the previous one failed, why, how, and when.

Raise your hand if you send alerts to Slack

Now keep it up if you're kind of embarrassed about that

Speaking of making sure someone knows the actual state of things… raise your hand if you send alerts to Slack. <give a moment, comment on number> <click> Now, keep it up if you're kind of embarrassed about that. <comment on how many go down>
We know alerts to slack (or similar apps, or email) is an anti-pattern. There's multiple reasons for that, but *one* of the reasons that alerts to your chat are not a great option is the lack of explicit feedback. You can't tell for sure that anyone has seen the notification, whether anyone has taken responsibility for responding to it, or whether the problem has stopped.

You know how this goes wrong…

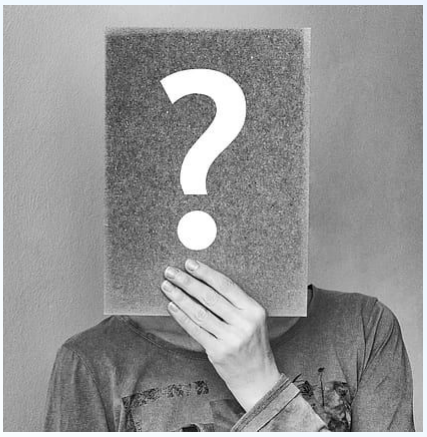


I've already given one kind of example on how alerts to Slack break the feedback loop and cause an outage. Here's another classic: we have a team who are responsible for making sure that various SSL certificates get renewed. It doesn't happen often so they hadn't gotten to automating it yet. One fairly important cert was about to expire, and an alert was posted to the team's slack channel reminding them to renew it. Several folks saw the notice… and assumed someone else was taking care of it (actually, in an especially classic version of this pattern, the person who usually did that was on vacation). The certificate was allowed to expire, causing an outage. Again, I have a specific example of this outage… but I bet you do too.

certificate expired image from
https://www.flickr.com/photos/danielbowen/6542901057

Fire-and-forget pages, too!

What if Email *is* Pages?

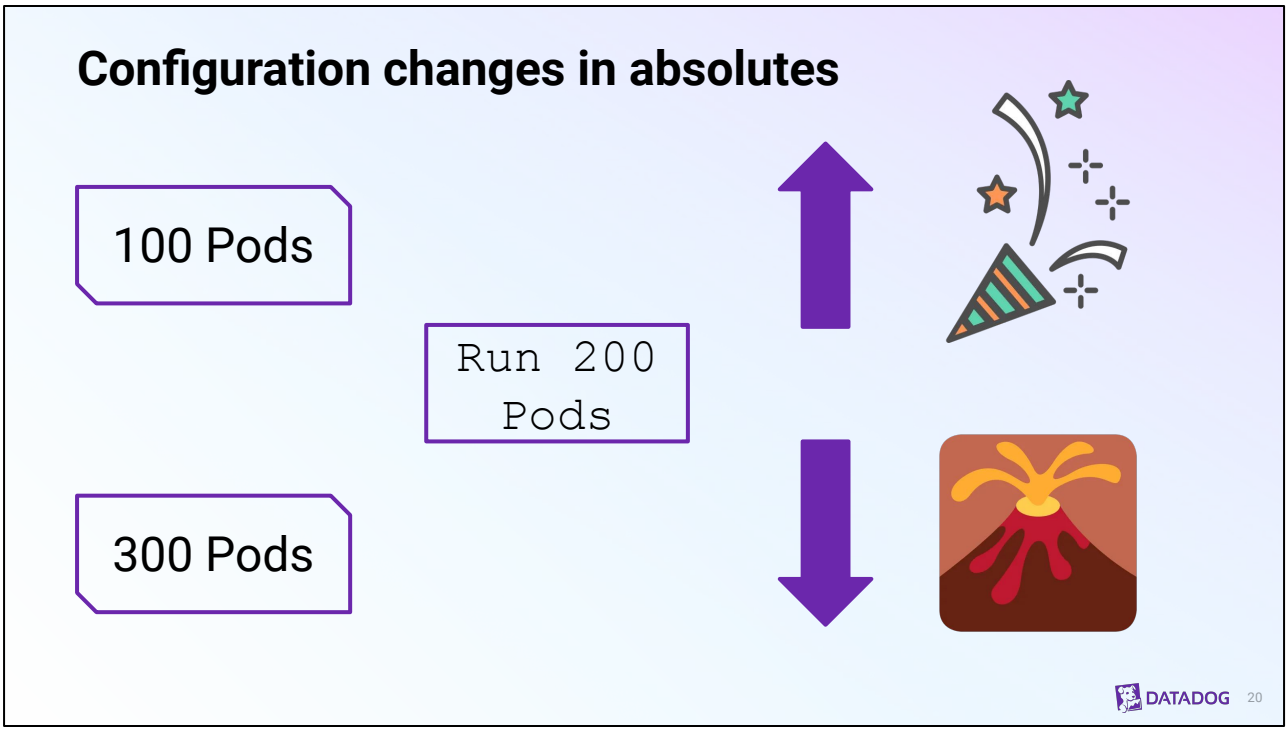


There's other ways your alerting system can lead you astray with not enough feedback. In my past life, I worked at a place where the standard way to page someone manually was by emailing their pager alias. <click> This wasn't "alerts to email" – the email actually caused a real page in a paging system to be created, and made a phone make noise. We were just using email as the UI.
But when there was an email outage, the mail system responded exactly as it should (accepting emails to send later, once it was back up, without delivering them).... and no one realized that pages weren't actually being sent. Luckily the outage was during our main business hours and teams were able to get ahold of each other on chat once they realized there was an issue, but the lack of feedback in the UI was a surprising challenge!

? photo from
https://www.pickpik.com/question-question-mark-survey-problem-test-solution-34684
Pager icon from Wannapik: https://www.wannapik.com/vectors/65890#!

And the right way to do better

You know the right answer here – alerts go to a pager, or a ticket queueing system that tracks them. And whatever UX you use to page people, it should come as close as possible to telling you whether the page was *actually delivered* as part of the sending function.



Here's another fun one – configuration systems that take values in absolutes *or* values in delta and aren't clear which one (or what you're doing). Personally I'm usually in situations where I intend to add N machines (or other resources) to a system, but I have to give the "number of machines to run on" as a total number. That means I have to do that simple arithmetic myself, and routinely the control system doesn't validate with me what the change I've made is (so it tells me "okay, I'm going to run 200 instances" but not whether thats more or less than the previous configuration). And this breaks in the obvious way – responders expect to upscale, but instead accidentally downscale their service (or, occasionally vice versa). Responders might get their arithmetic wrong (say they forget whether they're affecting a system replicated across multiple zones, or just they're under stress), or they might just not realize that the UI is in absolutes instead of a delta, accidentally downscaling their system.

Volcano icon from https://iconscout.com/free-icon/volcano-eruption-mountain-natural-disaster-emoj-symbol

Simple feedback!

100 Pods

Run 200 Pods

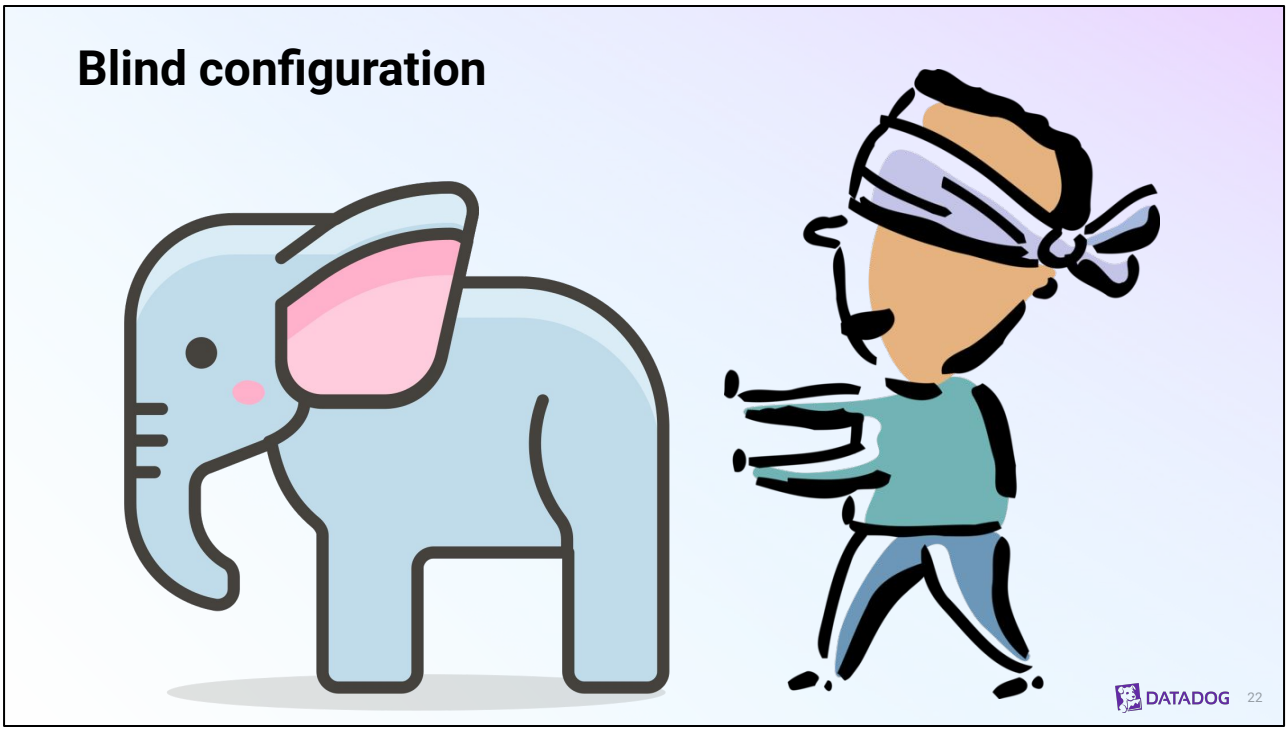
300 Pods

This will add 100 pods. Okay?

This will remove 100 pods. Okay?

It's so simple to fix! Have a confirmation that tell you the actual change being made – just a message saying "you're about to change from 200 instances to 300, is that right?" before actually taking the action.

Volcano icon from https://iconscout.com/free-icon/volcano-eruption-mountain-natural-disaster-emoj-symbol



Really there's a more general pattern here – any configuration change that doesn't show you the *actual impact* of that change before applying it. Many systems confirm the change being made, but they do it by just re-describing the change as you've already input it, rather than showing you the effect your change will have on production.

Blindfold Image from https://www.wannapik.com/vectors/67423
Elephant image from https://commons.wikimedia.org/wiki/File:469-elephant.svg

You don’t n**d th* l*tt*r ‘e’, right?

- Product r*dacts logs bas*d on r*g*x
- Us*r adds patt*rn r*placing ‘e’ for all logs in production
- No confirmation of numb*r of logs chang*d or *xampl* of *ff*cts

This one is easiest to explain with an example. So, we have a product that is designed to scrub sensitive data from logs, by matching patterns and redacting them if found. A user accidentally added a pattern that redacted the letter ‘e’ (replaced with ‘*’) from all logs in production. They were allowed to do this by the UI without a confirmation of how many logs were expected to be affected, or having to approve some example message changes. Confirming “you want to turn on xyz redaction rule” would have been better than nothing, but the *right* confirmation message to show here is “you are about to affect xx thousand logs per hour; here are some example impacted logs. Is this correct?”.
I’ll note that there’s other useful actions from this outage – certainly we should (and have) restricted the number of users who have access to make global changes in this part of the product, and implement better restrictions to prevent testing in production. We prevent outages with defense in depth, not just single confirmation dialogs – but feedback would have prevented this outage all on its own.

Show the results!

- You are about to affect 12345 pods, are you sure?
  - 7890 pods will have value foo changed to False
- This change is estimated to produce xxx more (or less) logs per hour. Make change?
- Row update will alter 5678 rows. Proceed?
- This IAM change will remove the bar permission for 255 users, including you. Proceed?

In general, users need to know the impact of their change – how many machines, or users, or logs, or database records are they about to affect? Above some reasonable threshold, *require a confirmation*. Give direct diff examples in the confirmation prompt if at all possible (old log vs. redacted log; user permission diff; etc). Some helpful tips:

- what *values* are changing
- avoid alert fatigue – don't confirm "normal", reasonable changes – only ones you expect to be dangerous, for example because they're especially large
- are you going to affect the current user?
- what about particularly important accounts (at DD, the production account for our own monitoring would be important to check)
- yes, this requires thought and judgment

“Location dependent” control planes

```
> % kubectl config set-context --current
--cluster=the_production_cluster
Context "current-context" modified.
> % thing
> % stuff
> % other work
> % etc
```


I’m gonna rag on the k8s CLI again for a minute here. K8s has this “helpful” feature where you can set the context, for instance the cluster that you’re taking operations in. And the result here is that you take actions in that cluster without having it front and center where you’re taking those actions. A lot of CLI tools have this property – they set a “location” or a context, and then any action you take happens silently in that context.

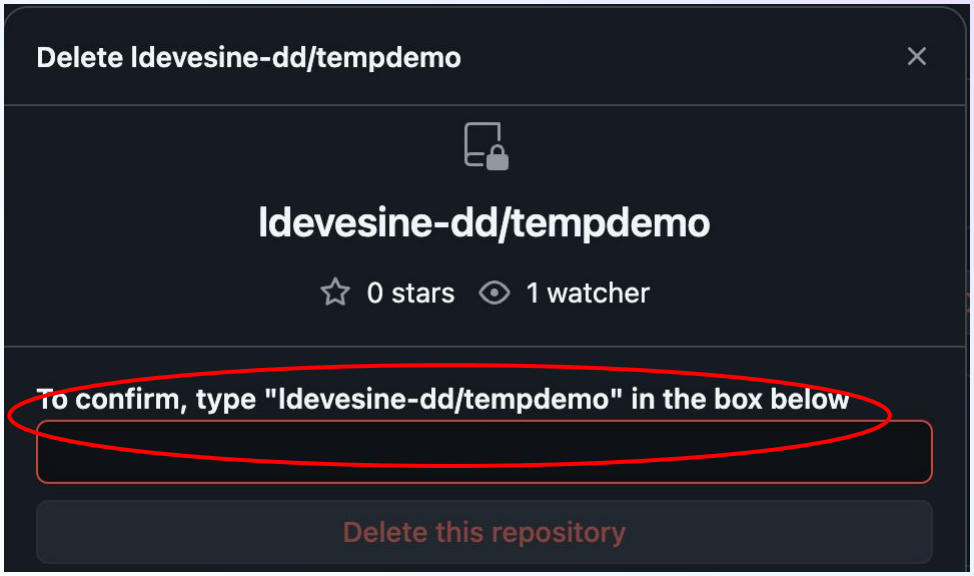
Which k8s cluster is in my context again?

```
> % foo
> % bar
> % baz
...
> % kubectl scale deployment --all --replicas=0
```


Stop me if you've heard this one! An engineer was doing some work in a personal sandbox cluster, wrapped up that work, and scaled the cluster to 0… except the cluster that they actually scaled to 0 was a production cluster (and then there was an outage!). Another example in the same vein is deleting the "test" database instance… except the engineer was actually logged into the production DB! This outage can haunt us anywhere that lets you set a context, and then take actions (especially destructive actions) without making sure you have situational awareness of that context.

Again this is a case where there's more than one solution worth applying! Reducing the number of engineers with permissions to scale the production cluster to 0 is certainly a Very Good Idea – but some folks will still need *some* level of control of production, and accidents happen. We're only human

Destruction requires situational awareness



We often include confirmation dialogs for destructive actions (“do you really want to delete the database? if so, type “really delete the database”) but forget to include a “situational awareness” check. Github actually does a fantastic job with this – to delete a repository, you need to type out the name of the repository you’re trying to delete. It’s valuable to go even further and include the “actual effect” of the change as well – “this repository has N contributors and xyz lines of code”; “there are nnn nodes and ppp pods currently running in this cluster that will be deleted”, “you are about to delete tt tables containing rr rows”. It’s not necessary to show confirmations of scale all the time, but it’s particularly a good idea past a threshold (users are generally less likely to want to take an action when there’s more data/users/scale effected).

Unvalidated configuration

```
scaling:
  cpu:
    enabled: true
    crossClusterBalancing: true
  minReplicas: 1
  maxReplicas: 200 # should be fewer than total partitions
```


Validation is really valuable in general – and unvalidated configuration is one of the most common ways we cause instant global outages as an industry. This can take a couple of forms – configurations that are allowed to ship to production, but are corrupted or unparseable, as well as configurations that *shouldn't* parse – they fail to define required fields, define fields that conflict with each other, or accidentally use a default that isn't safe for the system. A good warning sign <click> that you might have dangerous configurations is comments in your code being the only enforcement mechanism for not setting dangerous values, maybe that conflict with values elsewhere in code, rather than having an actual parser for the configuration enforcing that safety.

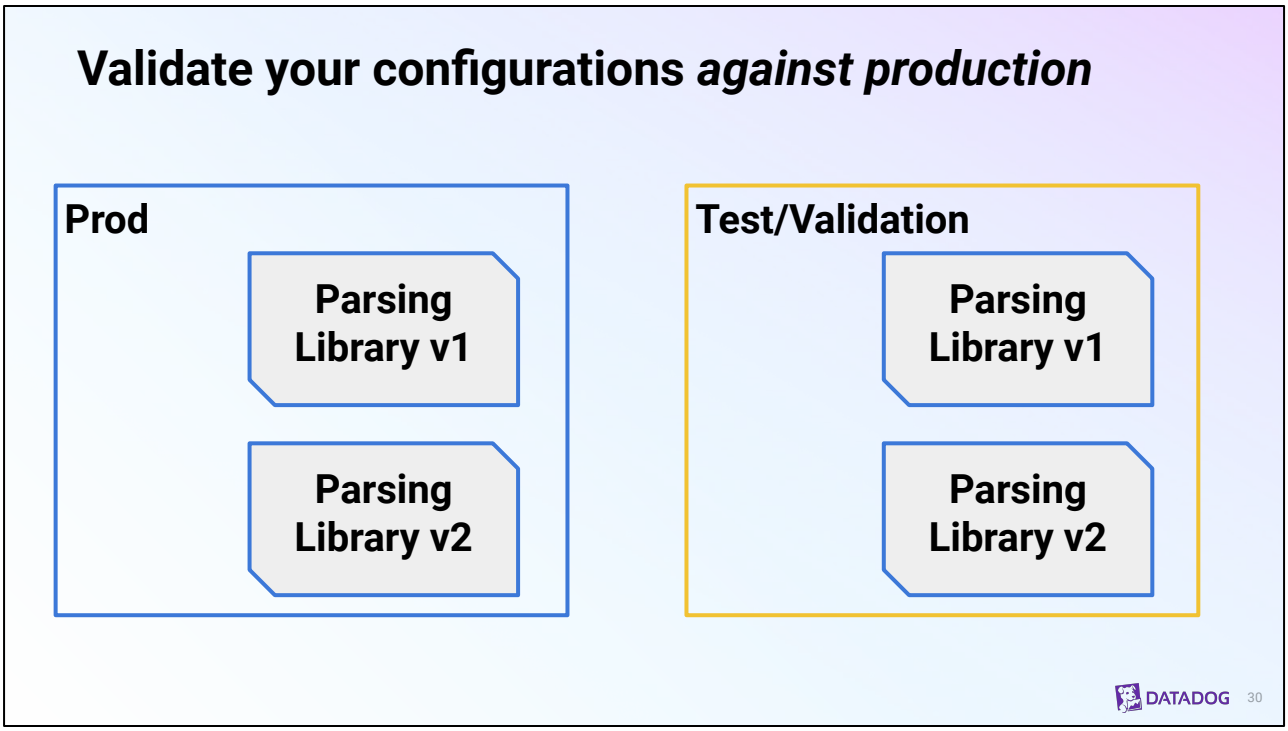
Crowdstrike

"On July 19, 2024, a Rapid Response Content update was delivered to certain Windows hosts, evolving the new capability first released in February 2024. The sensor expected 20 input fields, while the update provided 21 input fields."

- *Crowdstrike incident postmortem*

I was originally going to talk about a different example outage with configuration validation, but let's talk about crowdstrike!
So for those of you who live under a rock, on July 19 of this year Crowdstrike released a security rule (a kind of configuration) ~instantly globally that caused any Windows machine running Crowdstrike to crash and stay broken until manually fixed. They've released a full postmortem, but the short version is that the system validating the new configuration file was doing so using *different parsing code* than the system that actually read and used the configuration. This meant that the validator saw 21 "rule parameters" and marked it as valid, but when the interpreter on the customer machine actually tried to access parameter 21, it read memory out of bounds because it was only expecting 20 parameters.
This outage was quite bad for reasons besides the configuration parsing error – the practice of instant global rollouts for new configuration (which by the way is generally fairly standard in enterprise security products…) and the fact that the configuration parsing was happening during kernel startup hugely expanded the blast radius and impact of the issue.

Crowdstrike PM quote from
https://www.crowdstrike.com/falcon-content-update-remediation-and-guidance-hub/



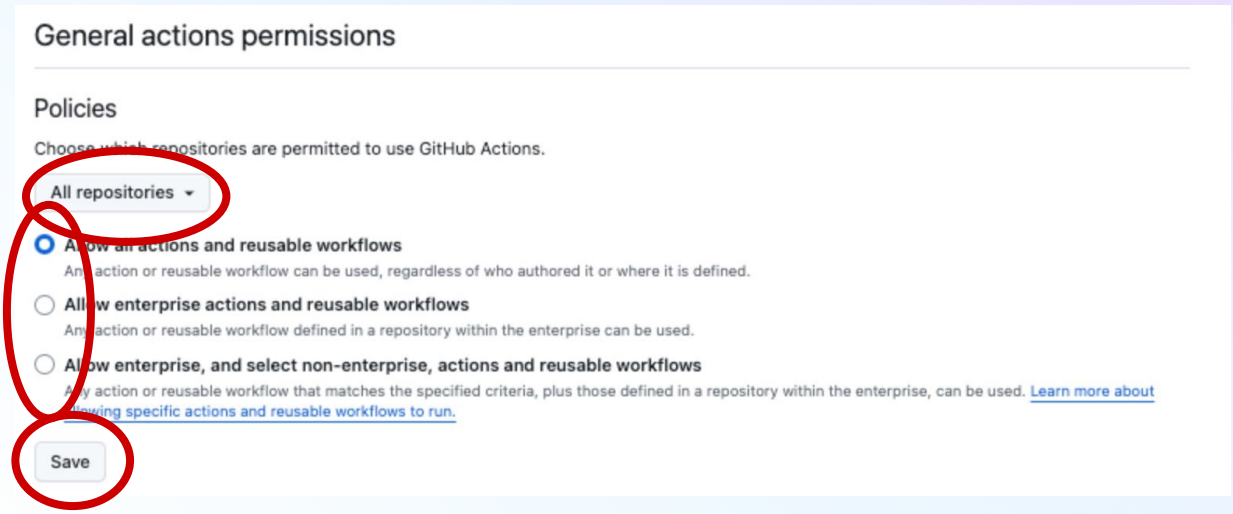
Configurations need to be validated by the actual parsing code that runs in production. If there are multiple versions of that parsing code (say because not every service is using the latest version), you need to validate against *every version running*. And because our configurations are frequently complex, with various layers of definitions and overrides, that validation should once again give feedback to your users – “hey, when I run this configuration vs. the previous one, here’s the delta of what services are *actually doing* so that you can confirm you have the effect you expect”.

Bad GUIs

GUIs are great for feedback, and I'm a huge fan of them – I can see a lot more information about what I'm doing on the screen in front of me, and I can press buttons instead of having to remember what the command is supposed to be to do whatever operation I want. But y'all, we write some *terrible* GUIs for ourselves and our engineer users. They can be misleading (confusing color coding or bad graph scales), overwhelm us with unimportant information, and they can make production changes that you don't expect them to.

Maze image from open clipart https://openclipart.org/detail/303427/distorted-maze

An example from Github



So, here's a section from Github's admin UI. I know it's a little hard to read, but this lets you control whether and how Github Actions are enabled across your org. There's three elements you can interact with on this page <click> this dropdown letting you select what repositories actions are enabled on, <click> these radio buttons describing what kinds of actions are enabled, and <click> a save button. <click>. So we had an engineer who was looking at our actions policies and opened up that dropdown to see what control actions were available, reported back what they'd learned, and went on with their day. Except that while they were doing that they'd actually selected a different option from the dropdown. And here's a *really* fun fact: that dropdown takes effect immediately, whithout touching that save button. The save button, it turns out, only applies to the radio buttons. And as a result, the engineer accidentally and unknowingly turned off Github actions for the entire Github org. Again, there's more than one thing to fix here – after this incident we significantly improved our monitoring on the current status and state of github actions, so someone gets a page if they get turned off again. We already have very limited access to this UI, so that doesn't help in this case. Unfortunately we can't fix the UI in Github ourselves, but we've definitely raised the problem.

User testing and designers

"**Hallway testing**, also known as **guerrilla usability**, is a quick and cheap method of usability testing in which people — such as those passing by in the hallway — are asked to try using the product or service."

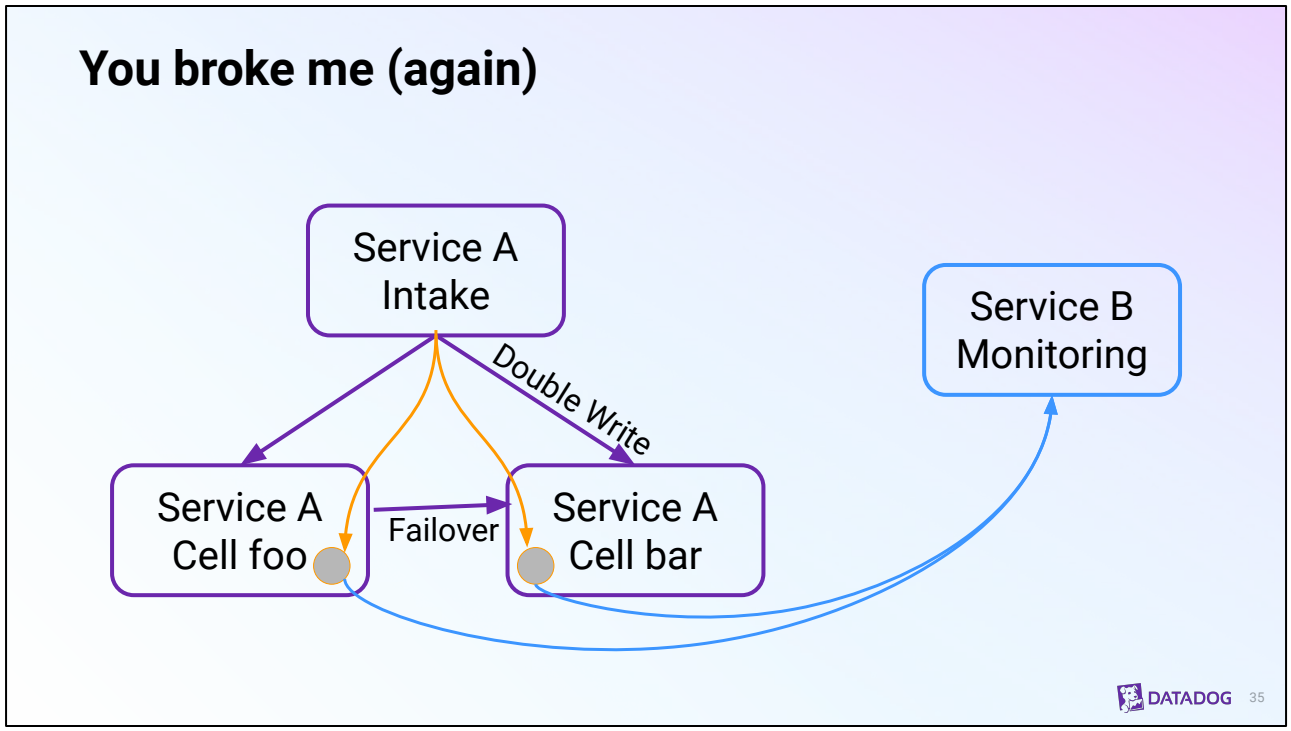
- Wikipedia (emphasis original)

Confirming "here's the configuration change you're making and the volume of things you're about to effect" help a lot in a UI. In addition, though, take the time, even for your "very simple" UI, to get feedback from someone who didn't write it on what their expectations are for how it should work and be used. If you can, *please* work with an actual designer – these folks have a *ton* of expertise on what leads to problems in a UI, and we should be willing to take advantage of their knowledge.

Wikipedia quote: https://en.wikipedia.org/wiki/Usability_testing on 2024-10-23

This final one is a little more subtle – it's the result of the fact that all of our systems are deeply interconnected. Because of that connectivity, it's often the case that making control plane changes to system A can cause serious issues for system B, owned by a whole other team. If team A doesn't have feedback on the state of system B in their own control plane, we can have serious outages that are entirely self-inflicted.

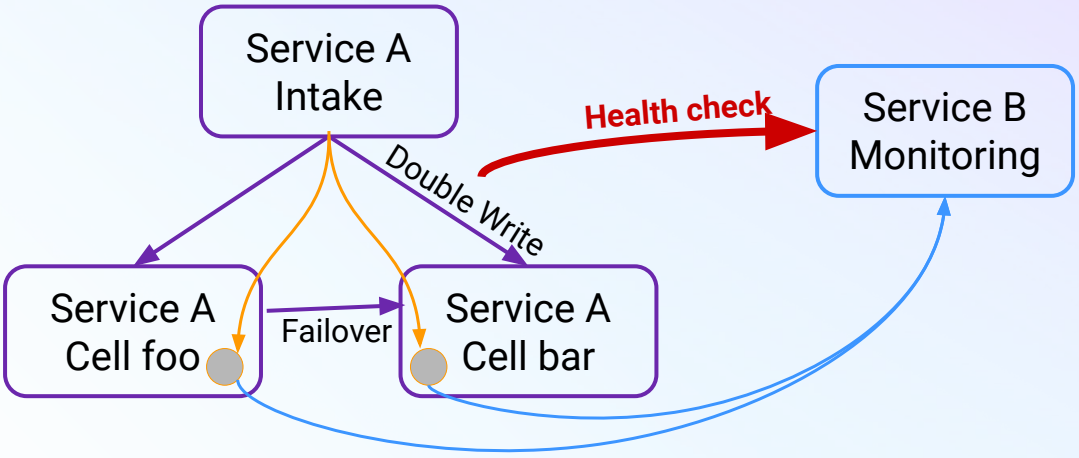
Spooky ghosts from https://pxhere.com/en/photo/791229



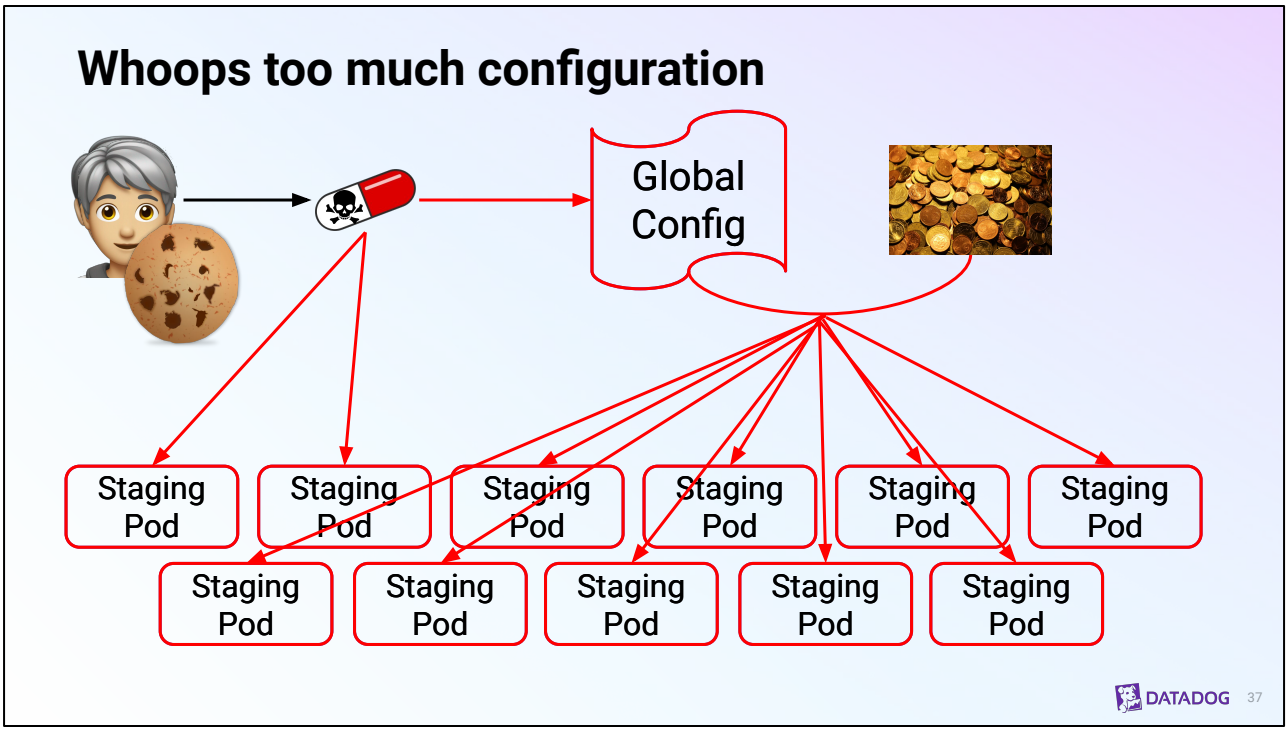
Here's an example of how that can happen. Team A has a cell-based structure for their system, where the intake normally writes to a single cell, let's say foo. The intake also <click> generates a kind of dialtone, which we use to monitor the progress of data through the pipeline. These dialtones <click> are read by service B to make action decisions. When the team operating service A does a failover between cells <click>, they need to double-write <click> for a period of time to allow for continuity for downstream readers. That double writing also includes generating duplicate dialtones. <click> This means that during the failover, Service B will see twice as many dialtones <click> (because of the duplicated data).

During an incident, team A needed to perform a cell failover, which led in this case to a memory issue in Service B. Team B was paged and started scaling up, but ran into an issue with the command in the runbook as they did. Unfortunately teams A and B were not in communication about the failover, so team A continued scaling up the double-writing, eventually causing user impact. This is a known problem with the interactions between these two systems, so it's not great to have it cause a user-facing incident – but without feedback built into the scale-up for team A, we're relying on the two teams to remember to talk to each other if there's a problem instead of being able to rely on the right information being in front of operators in the moment.

Release gates don’t care about the org chart

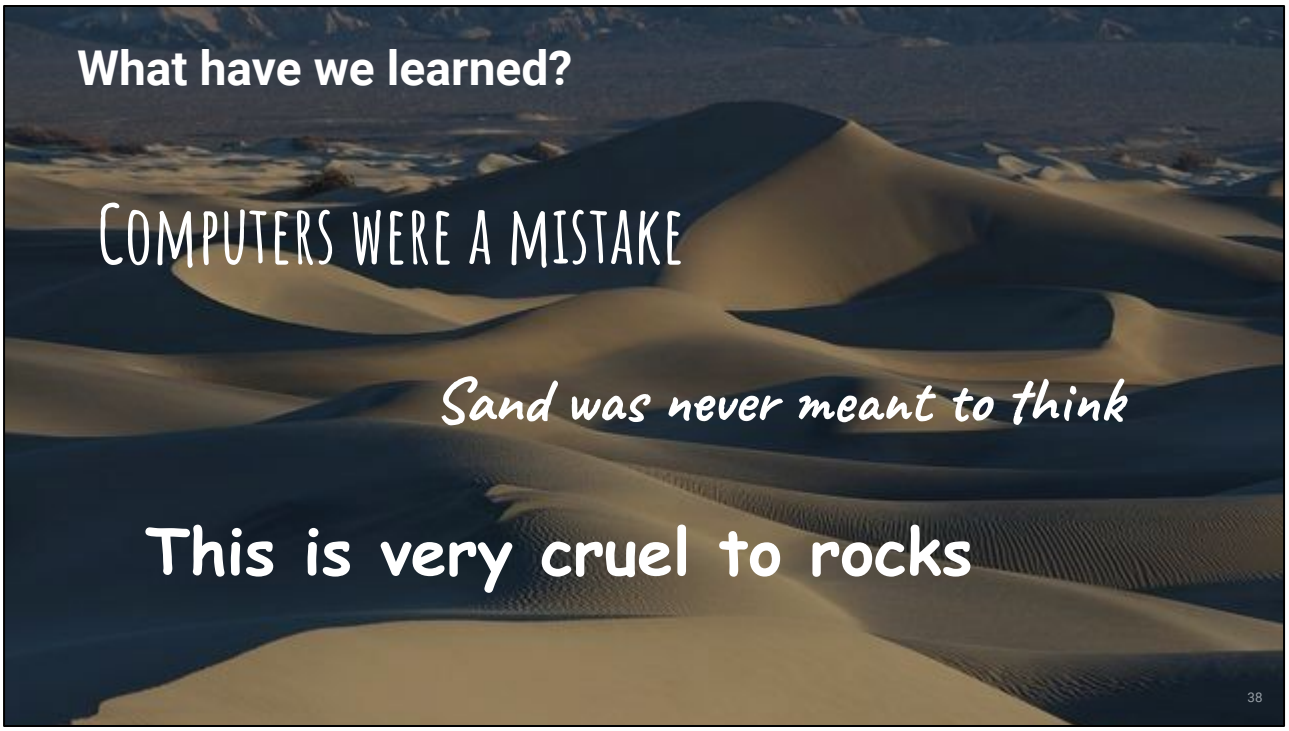


We can address this issue making sure that your release/change monitoring *doesn’t care about the org chart*. In this case, we updated automated release checks in team A’s system to specifically gate on the health of team *B*’s system. That means that team A will get direct feedback (and an automatic pause) to their failovers when team B’s system isn’t healthy.



“Spooky action” can take other forms as well, like accidentally changing configuration in unexpected ways. We had an experimenting with a live configuration system on our staging instance capable of changing running services immediately. Because of the work they were doing, they created a change that would (intentionally) crash any pods it was installed on <click>. The intern created this poison pill change and rolled it out to a couple of pods on staging, causing them to crash as expected. Unfortunately, <click< the configuration system was built to track the latest configuration globally, which meant that this change poisoned the global configuration as well <click> A bit later, someone else made an innocuous change <click>, and rolled it out globally… which also rolled out the crashing change globally <click>. And made our entire staging instance crash.
Just like the other spooky-action case, the solution here is to add release qualification gates in production that *don’t care about the org chart*. In addition to other config system changes, we specifically added gates here so that a new rollout will stop if the pods it’s installed on become unhealthy, regardless of who owns those pods or what the error actually looks like. And yes, we loudly praised the intern for finding this exciting outage in our config system! <click>

Intern image AI generated
Pill from openclipart (https://openclipart.org/)
Change image from
https://www.pickpik.com/money-coins-euro-coins-currency-euro-metal-34393



So, what have we learned here? Aside from the fact that computers are very hard and maybe forcing sand to think was a mistake, I think we can see that appropriate, timely, integrated user feedback in our control planes can make a big difference in preventing some of our outages. We're failing at this as a discipline right now – the fact that *so many* of us in this room have seen these *exact same* outages, or outages that closely rhyme, and we haven't fixed them yet has got to be a call to action. We can do such simple things to fix these! <next slide>

Show the actual change to production

- What is the ***end delta*** from current state
  - Scale of effect
  - Specific examples
- ***Where*** am I making the change
  - Do I mean to destroy my current context?
- What is the ***current status*** of changes in flight
  - And the current state of production

- Show it all in places the user is ***already looking***

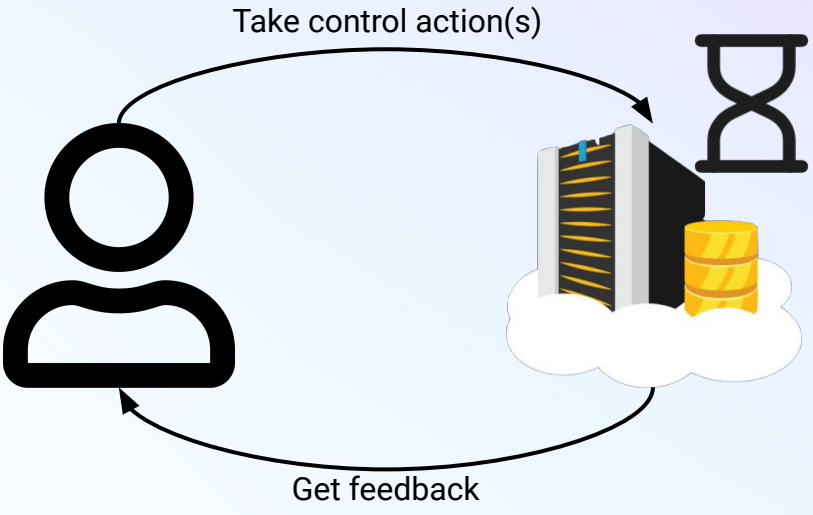
First and I think most importantly, you want to build feedback mechanisms at the point of use that show the *actual change* being made to production. <click> Show the user *how production will change*, including the delta from the current state, <click> the scope of the effect, <click> and examples of the change if appropriate and possible. If the change is large in scope, be sure to confirm it with the user with that context.
<click> Make sure the user knows where they are, <click> particularly for anything destructive (type the name of the repo!). <click> Report that production has changed, and <click> the *end results* of that change as they happen (the upscale has completed; the page has been delivered).
<click> and finally, don't force your user to *remember* to look for this data – let's offload that to robots, where it belongs, not our own precious brain meat.

Validate changes

Secondly, think about validation for changes. Always be looking at the *actual changes made* for validation – and always run the tests, have basic parsing and sanity checks for your configurations before they can be applied, and make sure those checks are the same as the actual parsing done in production (for goodness' sake use the same actual code!). Use a slow rollout and/or a canary with *team agnostic* release gates and pause and investigate if there's a problem.

Image public domain

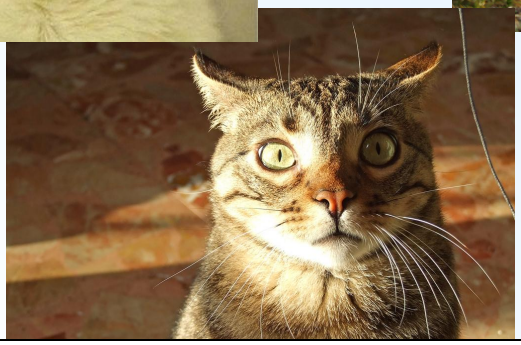
Keep the user informed



Even for slow or delayed changes, you need notifications and current-state feedback that is front and center for the user when they need it, *without* them asking for it (telling users to always check xyz before making a change to production is just asking humans to be better machines!). If there's a problem holding up a rollout, send that alert to somewhere it can't be missed – just like any alert your system generates.

User icon from Font Awesome Free 5.2.0 by @fontawesome - https://fontawesome.com
Server in cloud Image by https://www.vectorportal.com
hourglass icon from https://davooda.com/basic-icons/basic-time-hourglass-sand-timer-icon

Surprise is bad. Feedback is good.



Look for signs of surprise in your postmortems and incident follow ups, and think about how you can build informative, active feedback mechanisms that can prevent future outages. Even better, have a look at the tooling you're using today, and think about whether you're getting feedback from your tools on the effect you're actually having. A little feedback can prevent a lot of pages – the next time you sit down to write a control plane UX, even (or especially!) if it's "just a quick CLI" or "temporary", take a few minutes to build in a feedback loop. Future you will thank you, and we'll all be better SREs.

Surprised cats from https://www.flickr.com/photos/54125007@N08/15634745431, https://www.flickr.com/photos/gattomimmo/318700028

Thank You!
Questions?